Postel
9926

Network Working Group
Request for Comments #333
NIC #9926
Category: C9 (experimentation)
Obsoletes: 62
Updates: none

Bob Bressler, MIT/Dynamic Modeling
Dan Murphy, BBN/TENEX
Dave Walden, BBN/IMP

Rcvd
22 June 72

15 May 1972

# A PROPOSED EXPERIMENT WITH A MESSAGE SWITCHING PROTOCOL

CONTENTS


| | |
|---|---|
| Introduction | 1 |
| Some Background | 2 |
| References | 4 |
| MSP Specification | 5 |
| Issues | 13 |
| Message Header | 16 |
| Examples | 23 |
| TELNET | 25 |
| The Information Operator | 26 |
| Unique Port Numbers | 32 |
| Flow Chart | 36 |
| MSP Variations | 40 |
| Appendix | |

## Introduction

A message switching protocol (MSP) is a system whose function is to switch messages among its ports.

For example, there is an implementation of an MSP in each Interface Message Processor. We believe that the effective utilization of communications networks by computer operating systems will require a better understanding of MSPs. In particular, we feel that Network Control Programs (NCPs), as they have been implemented on the ARPA Computer Network (ARPANET), do not adequately emphasize the communications aspects of networking—i.e., they reflect a certain reluctance on the part of systems people to move away from what we term "the stream orientation". We propose, as an aside to network development using the current NCPs, to rethink the design of NCP-level software beginning with a consideration of MSPs.

The thrust of this note is to sketch how one would organize the lowest level host-host protocol in the ARPANET around MSPs and how this organization would affect the implementation of host software.

Some Background

Over the past several weeks there has been considerable informal discussion about the possibility of implementing, on an experimental basis, in several of the ARPA Network Host Computers, NCPs which follow a protocol based on the concept of message switching rather than the concept of line switching (see the parenthetical sentence in the first paragraph of page 6 of NIC document 8246, Host/Host Protocol for the ARPA Network). Party to this discussion have been Bob Bressler (MIT/Dynamic Modeling), Steve Crocker (ARPA), Will Crowther (BBN/IMP), Tom Knight (MIT/AI), Alex McKenzie (BBN/IMP), Bob Metcalfe (MIT/Dynamic Modeling), Dan Murphy (BBN/TENEX), Jon Postel (UCLA/NMC), and Dave Walden (BBN/IMP).

Several interesting points and conclusions have been made during this discussion:

1. Bressler has implemented a message switched interprocess communication system for the Dynamic Modeling PDP-10 and has extended it so it could be used for interprocess communication between processes in the Dynamic Modeling PDP-10 and the AI PDP-10. He reports that it is something like an order of magnitude smaller than his NCP.

2. Murphy has noted that a Host/Host protocol based on message switching could be implemented experimentally and run in parallel with the real Host/Host protocol using some of the links set aside for experimentation. Further, Murphy has noted that if this experimental message switching protocol were implemented in TENEX, a number of (TENEX) sites could easily participate in the experiment.

3. It is the consensus of the discussants that Bressler should take a crack at specifying a message switching protocol* and that

---

*This note fulfills any obligation Bressler may have incurred to produce an MSP specification.

if this specification looked relatively easy to implement, a serious attempt should be made by Murphy and Bressler to find the resources to implement the experimental protocol on the two BBN TENEX and the MIT Dynamic Modeling and AI machines.

4. MSP was chosen as the acronym for Message Switching Protocol, and links 192-195 were reserved for use in an MSP experiment.

We solicit comments and suggestions from the Network Working Group with regard to this experiment. However, although we will very much appreciate comments and suggestions, because this is a limited experiment and not an attempt to specify a protocol to supersede the present Host/Host protocol for the ARPA Network, we may arbitrarily reject suggestions.

References

Familiarity with the following references will be helpful to the reading of the rest of this note.

1) NIC document 8246, Host/Host Protocol for the ARPA Network;

2) NIC document 9348 on the Telnet Protocol;

3) NIC document 7101, Official Initial Connection Protocol, Document #2;

4) A System of Interprocess Communication in a Resource Sharing Computer Network, CACM, April, 1972.

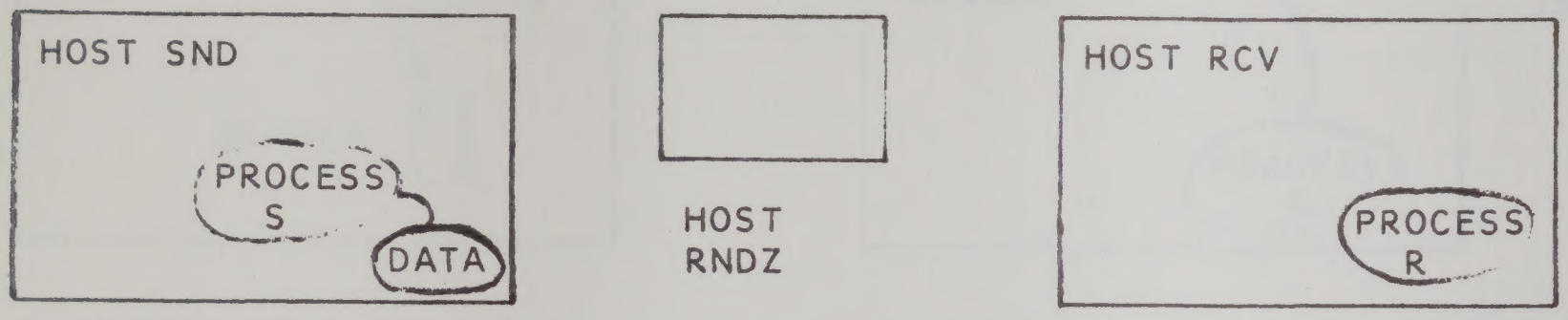
Reference 4 is a revision of RFC 62. We strongly suggest the reader be familiar with reference 4 before he attempts to read the present RFC; a reprint of reference 4 is attached as an appendix.

MSP Specification

Our MSP is essentially a generalization of the interprocess communication system outlined in Section 3 of the fourth reference. (Henceforth, if we are required to mention the interprocess communication system presented in Section 3 of reference 4, we shall call it "the IPC".) For two processes to communicate using the MSP, the process desiring to send must in some sense execute a SEND and the process desiring to receive must in some sense execute a RECEIVE. The SEND and RECEIVE, in effect, rendezvous somewhere and transmission is allowed to take place. With the RECEIVE are specified (among other things) a from-port-id, a to-port-id, and a rendezvous Host. With SEND are specified a from-port-id, a to-port-id, a rendezvous Host, and (possibly) some data to be transmitted. Using SEND and RECEIVE, sending a message from a sender process to a receiver process takes place as follows. The sender process executes a SEND which causes an OUT-message plus the specified data to be transmitted to the Host specified as the rendezvous Host in the SEND. Concurrently (although not necessarily simultaneously) the receiver process executes a RECEIVE which causes an IN-message to be sent to the Host specified as the rendezvous Host in the RECEIVE. At the rendezvous Host, OUT-messages and IN-messages are entered in a table called the rendezvous table. When an OUT-message and an IN-message are detected with matching to-port-id, from-port-id, and rendezvous Host, three things are done: 1) the OUT-message plus the data is forwarded to the Host which was the source of the IN-message, 2) the IN-message is forwarded to the Host which was the source of the OUT-message, and 3) the IN-message and OUT-message plus the data are deleted from the rendezvous table in the rendezvous Host.

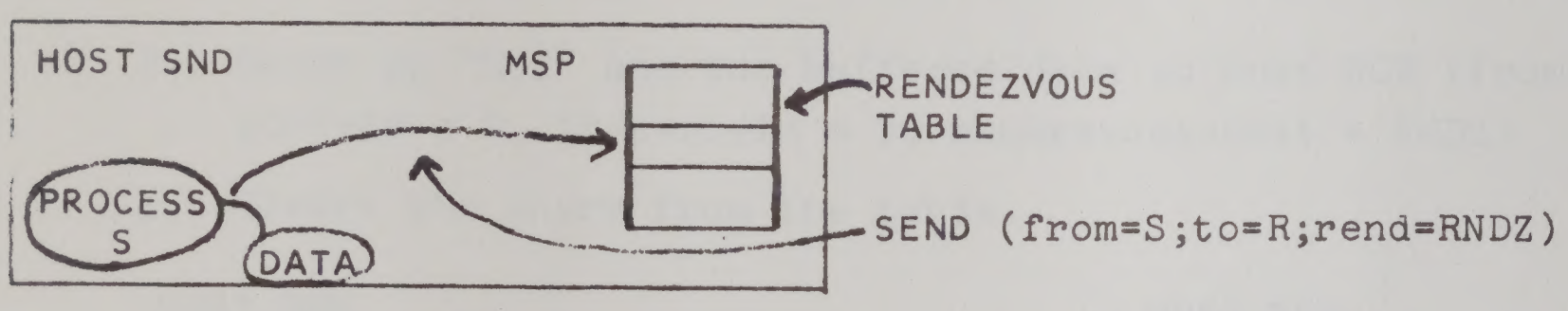
The process is greatly simplified if the rendezvous Host is also either the send Host or receive Host. Specific algorithms enumerating these sequences appear later in this note.

To clarify the basic concepts, let us look at a case involving three Hosts, to which we shall give the names SND, RCV, and RNDZ. At Host SND, process S is doing a send, and at Host RCV, process R is doing a receive. Both specify rendezvous at Host RNDZ.

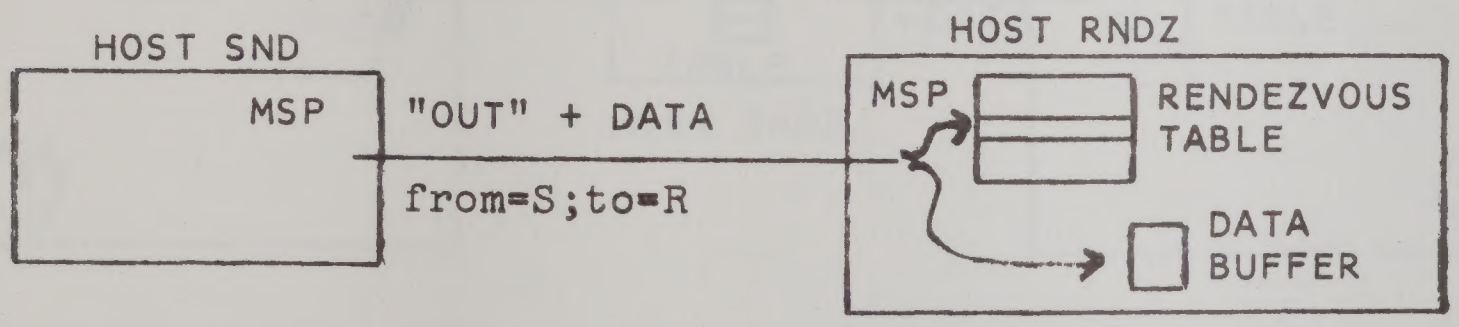



Process S now executes a SEND with

from-port-id = S, to-port-id = R, and rendezvous-Host = RNDZ. Host SND then creates a table entry in its rendezvous table.



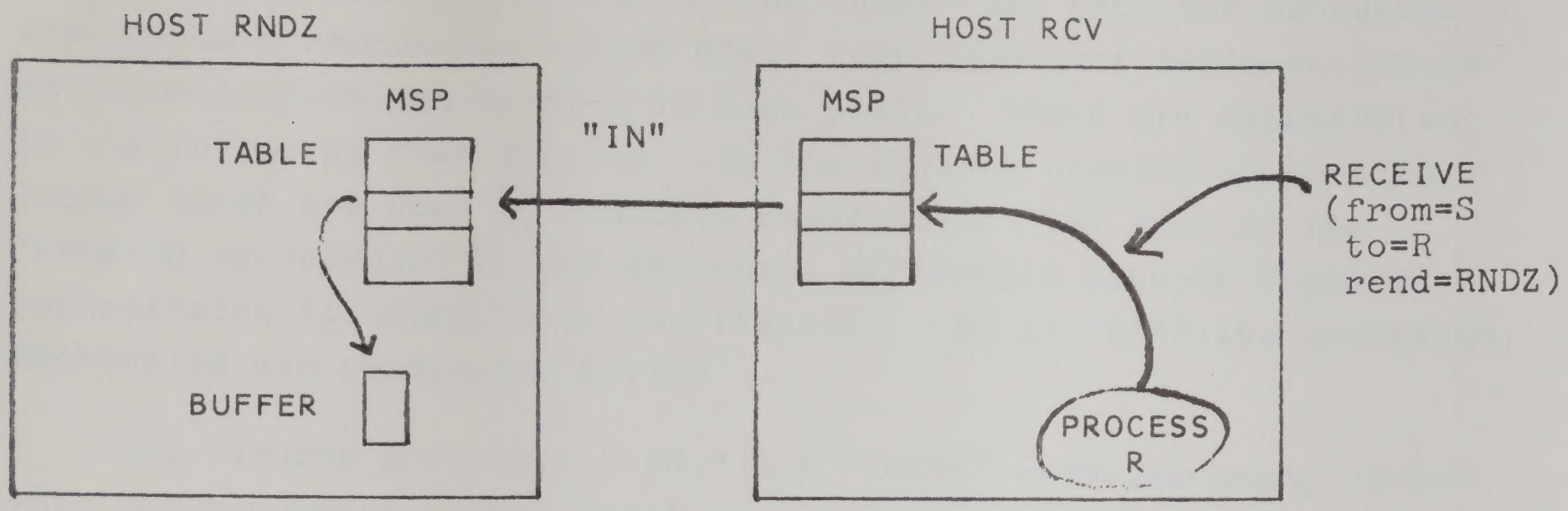

Host SND now sends an "OUT" message with S's data to Host RNDZ.




Concurrently process R at Host RCV executes a RECEIVE with from-port-id = S, to-port-id = R, and rendezvous-Host = RNDZ. As above, Host RCV creates a table entry in its rendezvous table and sends an "IN" message to Host RNDZ (see following figure).*

---

*Don't panic now about buffering in an intermediate Host. The time to panic is after you've read and understood the rest of our arguments.

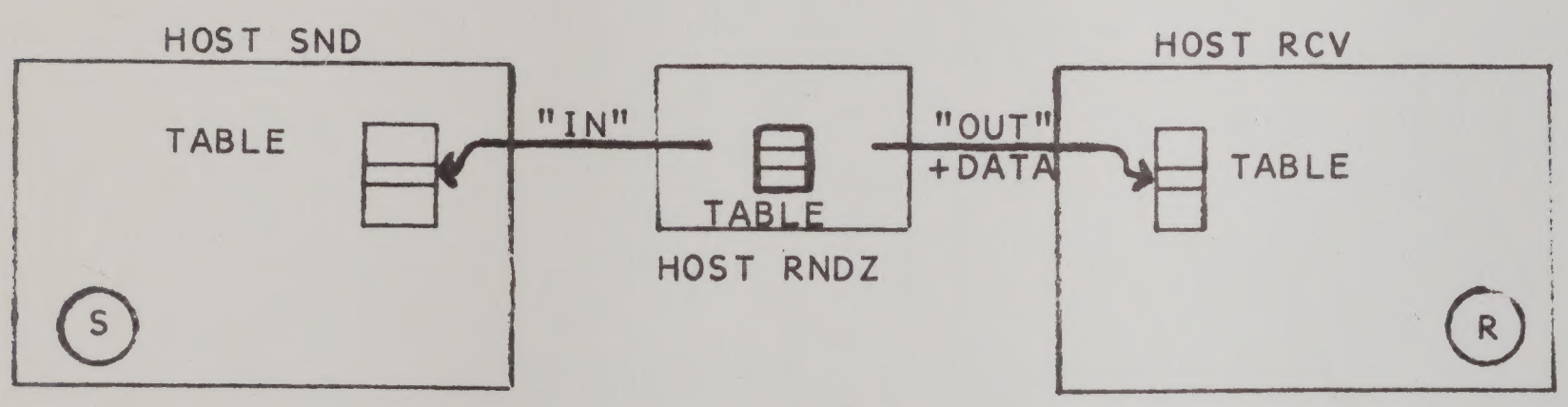



Host RNDZ now notices that the "OUT" from Host SND and the "IN" from R at RCV match one another and thus Host RNDZ takes three actions:

1. Sends an "IN to Host SND (from-port-id = S, to-port-id = R, rendezvous-Host = RNDZ).

2. Sends an "OUT" and the buffered data to Host RCV (from-port-id = S, to-port-id = R, rendezvous-Host = RNDZ)

3. Clears the entry from its table.



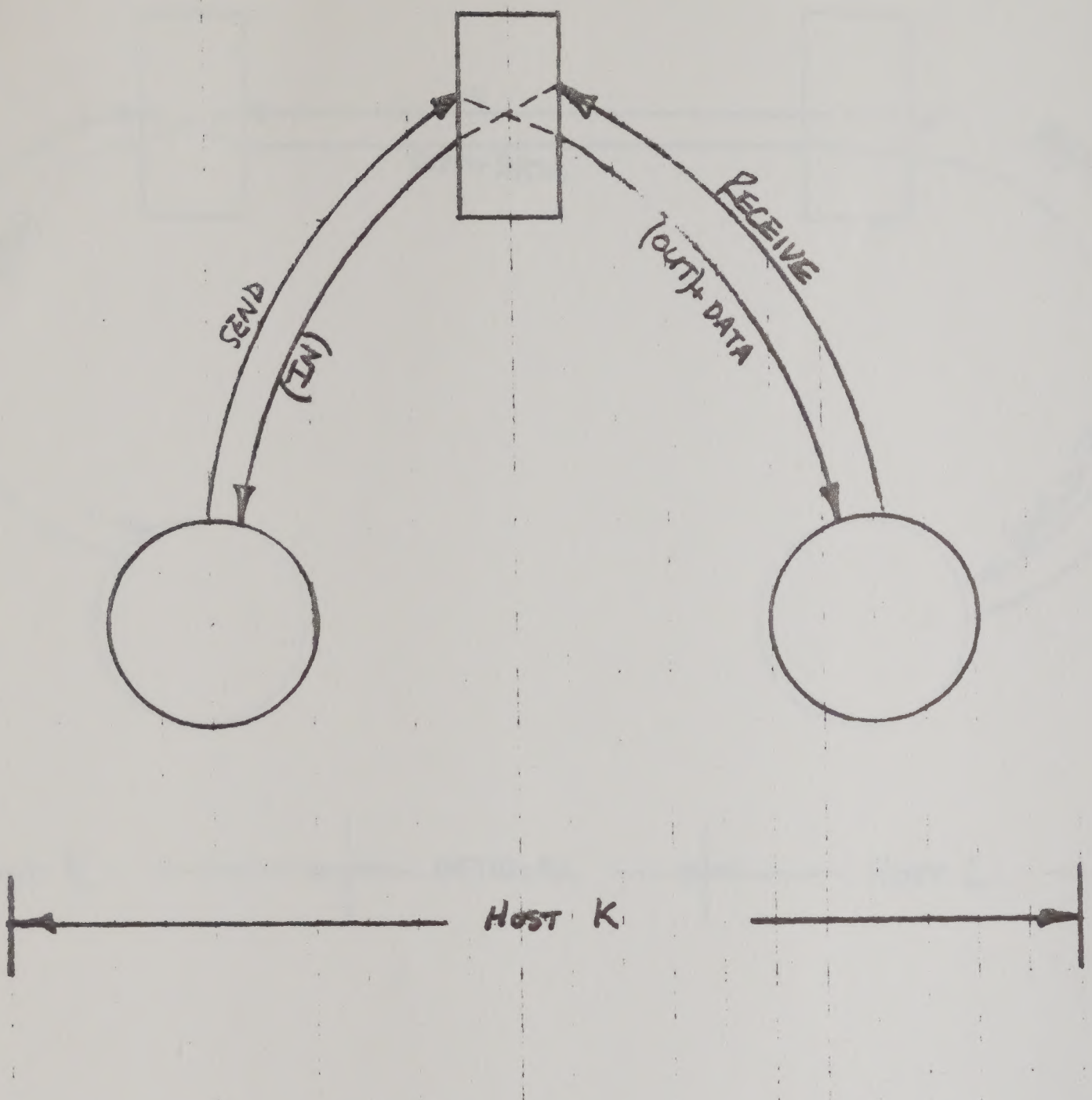

Host RCV gets the "OUT" and DATA and finds the matching entry in its table. It gives the DATA to process R and clears the entry from its table.

Host SND gets an "IN" which matches an entry in his table and clears that entry. This message serves as a combined acknowledgment and go ahead which can be passed along to process S.

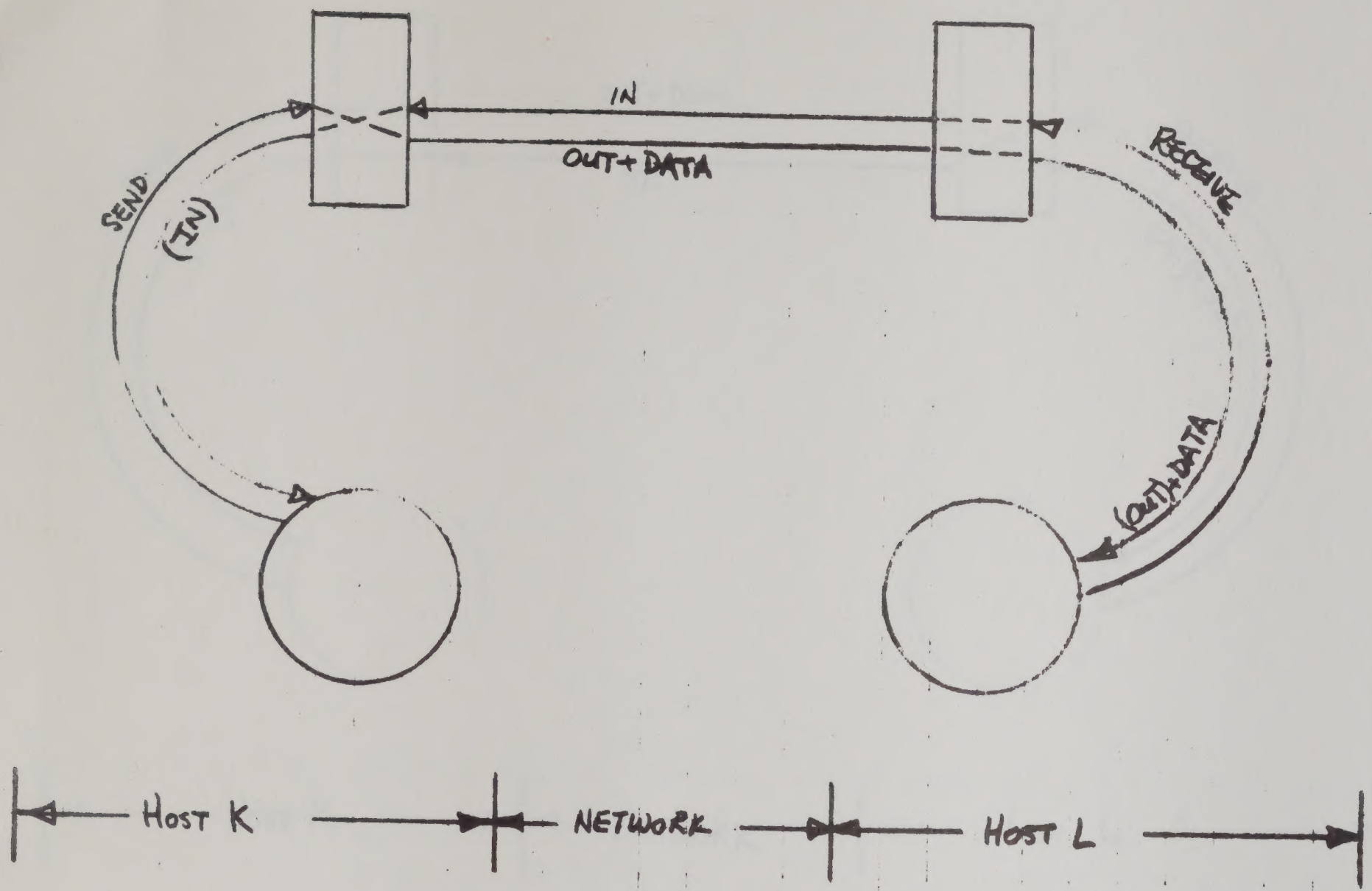
The transmission is now complete.

By both, one, or neither of the sender and receiver processes specifying a remote rendezvous Host, four important different kinds of transmissions can be made to take place. These are illustrated in the following four figures. In the figures crossed or parallel dotted lines are used to indicate rendezvous. The site of the "crossed rendezvous" is the important difference between types of transmission illustrated in the figures. Circles indicate processes. Rectangles are rendezvous tables.

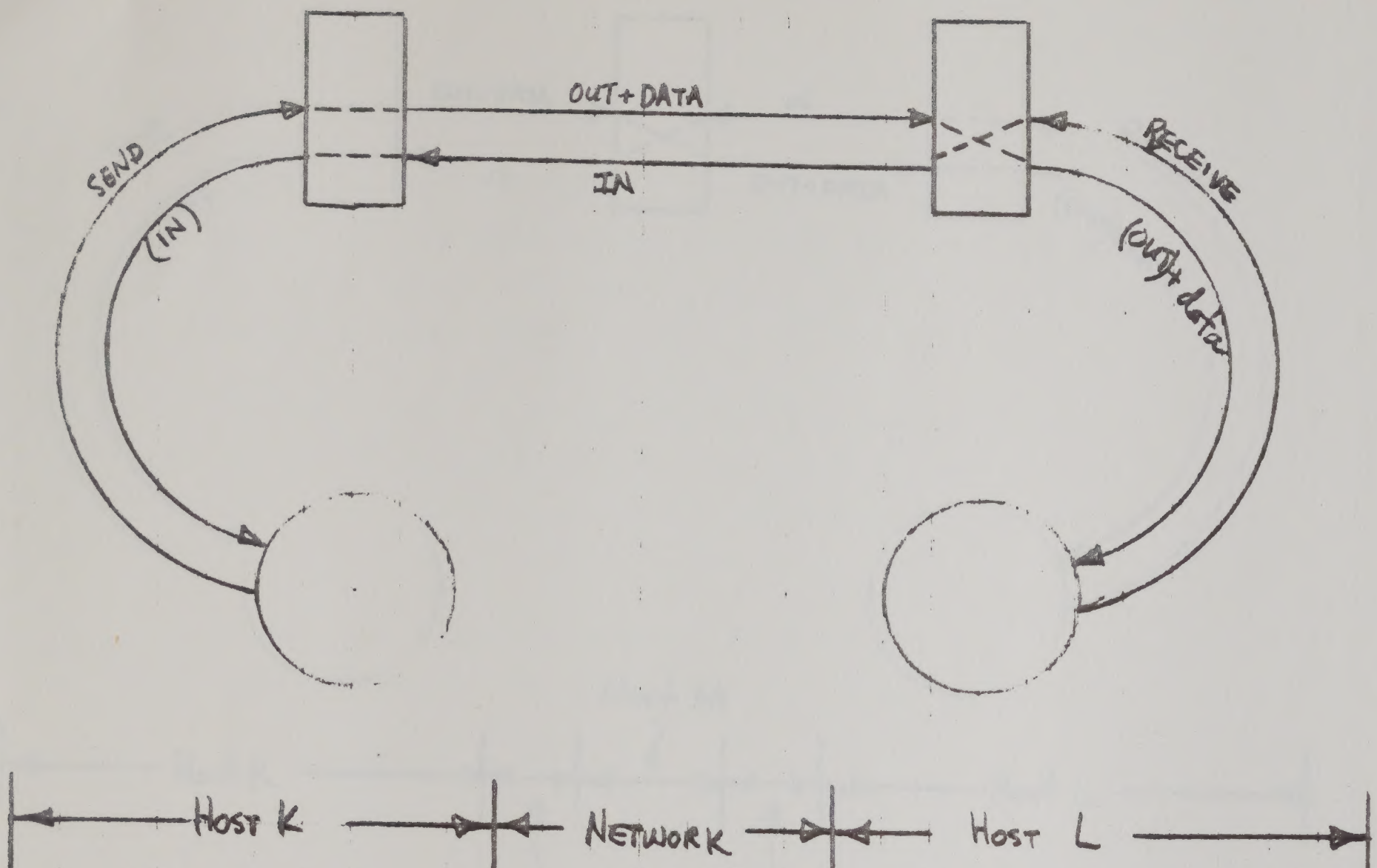
The figures also show "(IN)" and "(OUT)" messages being passed into the processes. The parentheses are used to indicate that the "IN" and "OUT" are only <u>conceptually</u> passed into the processes. What actually happens is implementation dependent. The process might be awakened and be given no further information if it blocked when issuing the SEND or RECEIVE. The process might be interrupted and passed some information such as the to-port-id from the IN or the from-port-id of the OUT. The process might actually be passed the complete IN or OUT message.



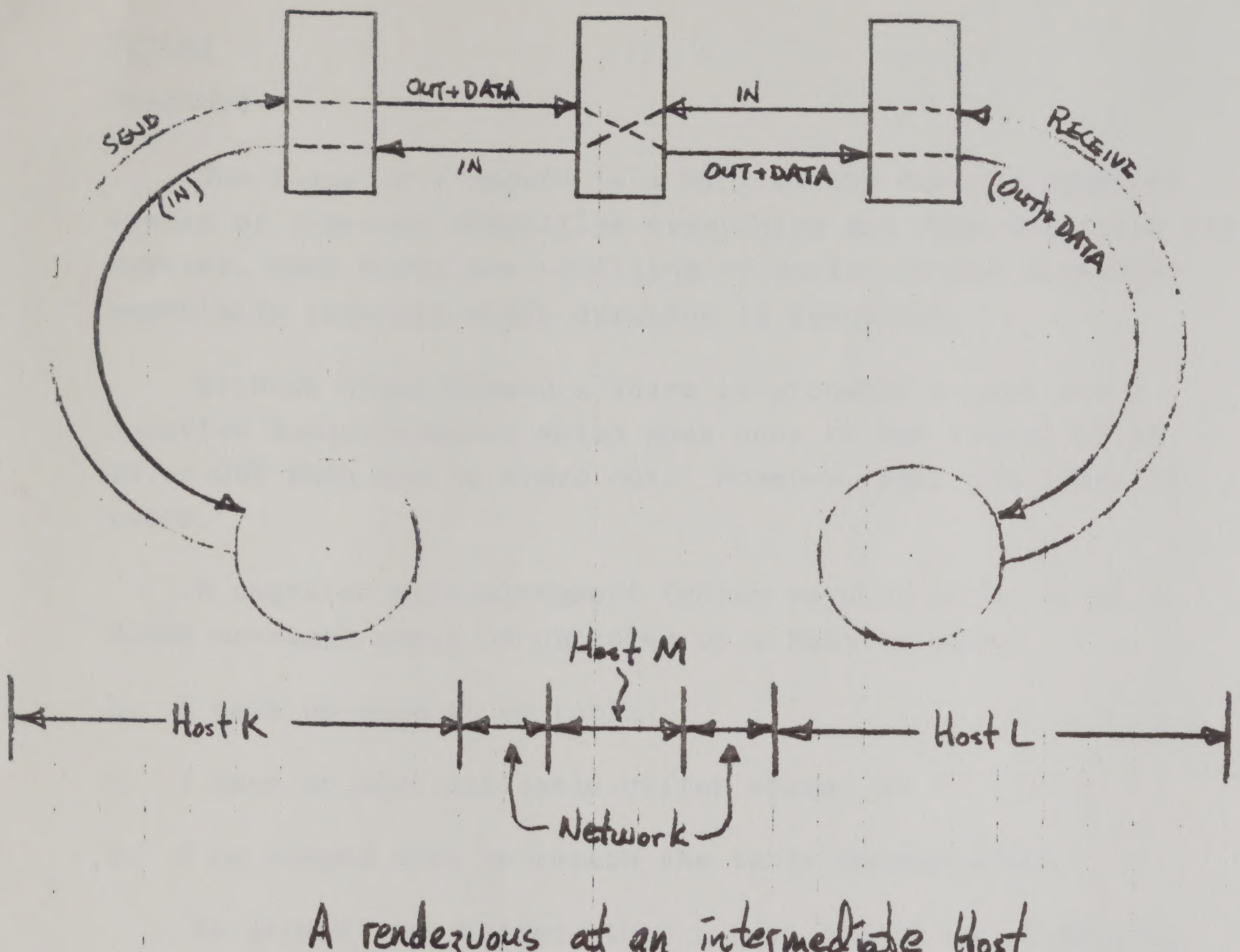

A completely local rendezvous




A rendezvous at the sender's Host




A rendezvous at the receiver's Host




A rendezvous at an intermediate Host

Issues

Timeouts.

The issue of timeouts is a very sticky one. A coherent system of timeouts simplifies everything and does away with races. However, many Hosts are unwilling or unable to use timeouts, especially timeouts whose duration is specified.

Without these timeouts there is probably a need for a negative acknowledgment which goes back to the source of an IN or OUT when one is timed out. However, this now leads to races.

A negative acknowledgment (which we will refer to as a FLUSH message) could be employed by a Host to mean:

1. I have no room in my table

2. I have no more available buffer space, or

3. I no longer wish to retain the table entry/buffer.

In general, we believe that a Host should be allowed to throw away an IN or OUT+data whenever it is no longer convenient for the Host to hold the messages. This can be immediately on the arrival of a message; for instance, if the Host does not want to buffer traffic for which it does not have a user buffer. In lieu of timeouts, any time a process issues a SEND or RECEIVE, it can take it back by issuing the matching RECEIVE or SEND.

Blocking the process after a SEND or RECEIVE.

This is a question which is left implementation dependent. In general, we do not think it is a good idea to block the process after a SEND since it may want to do another to another port or even do a RECEIVE. In fact, we see nothing inherently wrong with a process doing two or more SENDs to the same port as long as

the communicating processes know what they are doing. Of course, some communicating processes will prohibit several simultaneous messages being in transit between the same ports, for instance the TELNETs may well prohibit this. However, for reasons of increasing bandwidth, etc., two processes may well want several simultaneous messages. In this case we think it is up to the processes to worry about the sequencing of messages; however, we refer users desiring their processes to take care of message sequencing to the method used in the IMP/Very Distant Host interface which is documented in Appendix F of BBN Report 1822.

Message Buffering

A few points are worth mentioning with regard to message buffering. First, most OUTs will probably be accompanied by data. Therefore, in general, since the receiver process may be swapped out, the receiver Host monitor must be prepared to buffer some data somewhere. To minimize the amount of buffering needed, the monitor could refuse further traffic from the IMP until the earlier traffic from the IMP has been written on a disk or drum. Or the monitor could have a small number of buffers in the monitor area of memory which it fills as traffic comes from the IMP, and which are swapped with buffers claimed earlier by the receiver processes as the receiver processes are swapped in. Note that the buffers may be less than the maximum subnet message size in length if the RECEIVEs never specify a longer message length—of course, this can be enforced. Finally, note that the message size, receive-port-id, etc. are available in the first 144 bits which come in from the IMP. It might be useful to read this before deciding into which buffer to read the rest of the message.

Positive Acknowledgments

Built into the system is a certain form of acknowledgment. The information is always available as to when the receiving process has done a RECEIVE. The sending Host is assured of receiving an "IN" when the receive call is issued.

Further forms of acknowledgment and validation can be implemented at the first user level, and advanced protocols will probably develop a library of such routines.

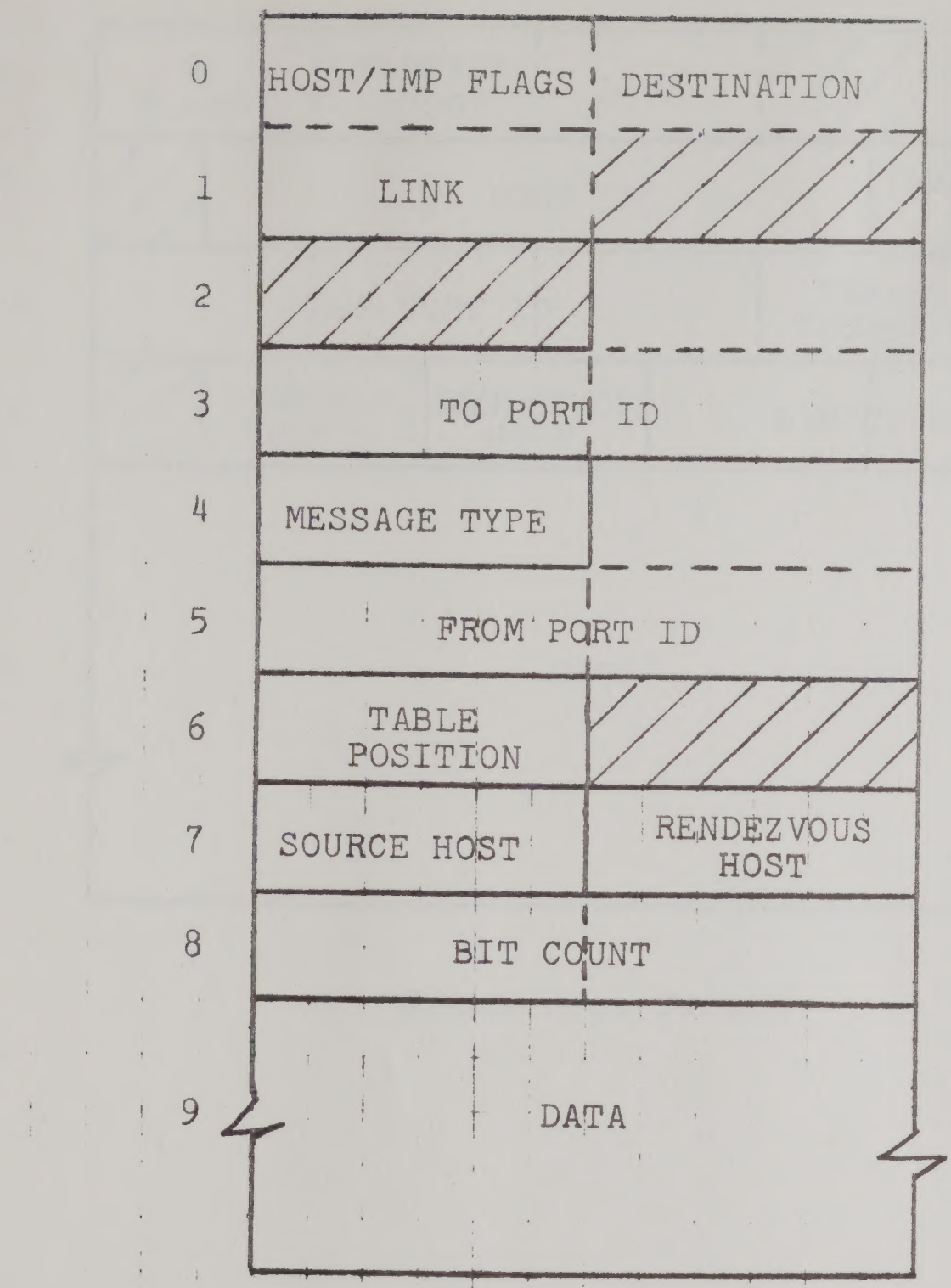
Message Header

The following section deals with the specific format of Host to Host messages and algorithms describing the proper response to a given message.

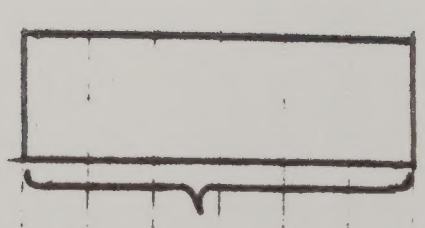
Each message begins with a 144 bit header containing the following fields:

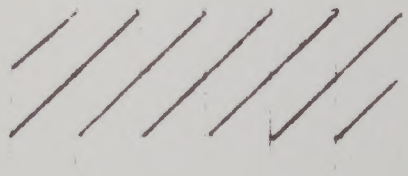
1. HOST-TO-IMP leader (32 bits) as specified in BBN Report 1822
2. to port ID (i.e., the id of the port receiving the message) (24 bits)
3. MSG TYPE (8 bits) IN, OUT, FLUSH, etc.
4. from port ID (i.e., id of the port sending the message) (24 bits)
5. initiating Host's table position (8 bits) see below.
6. HOST "sourcing" this message (8 bits) see below.
7. RENDEZVOUS HOST (8 bits)
8. bit count of data (16 bits)

The header format has been arranged so that no data item will cross a word boundary on machines with 16, 32, and 36-bit words, except where the size of the item is greater than the word size. The actual arrangement of bytes within words is shown in the following figures for these three word sizes. For the benefit of 36-bit Hosts, bytes 4 and 13 (numbering from Ø) are unused. The 2 and 3-byte items do not cross word boundaries except for the port ID's on the 16-bit machines. This attention to packing and unpacking ease was given both for general convenience, and in particular because Hosts may wish to examine the header at interrupt level to determine where the rest of the message should go.

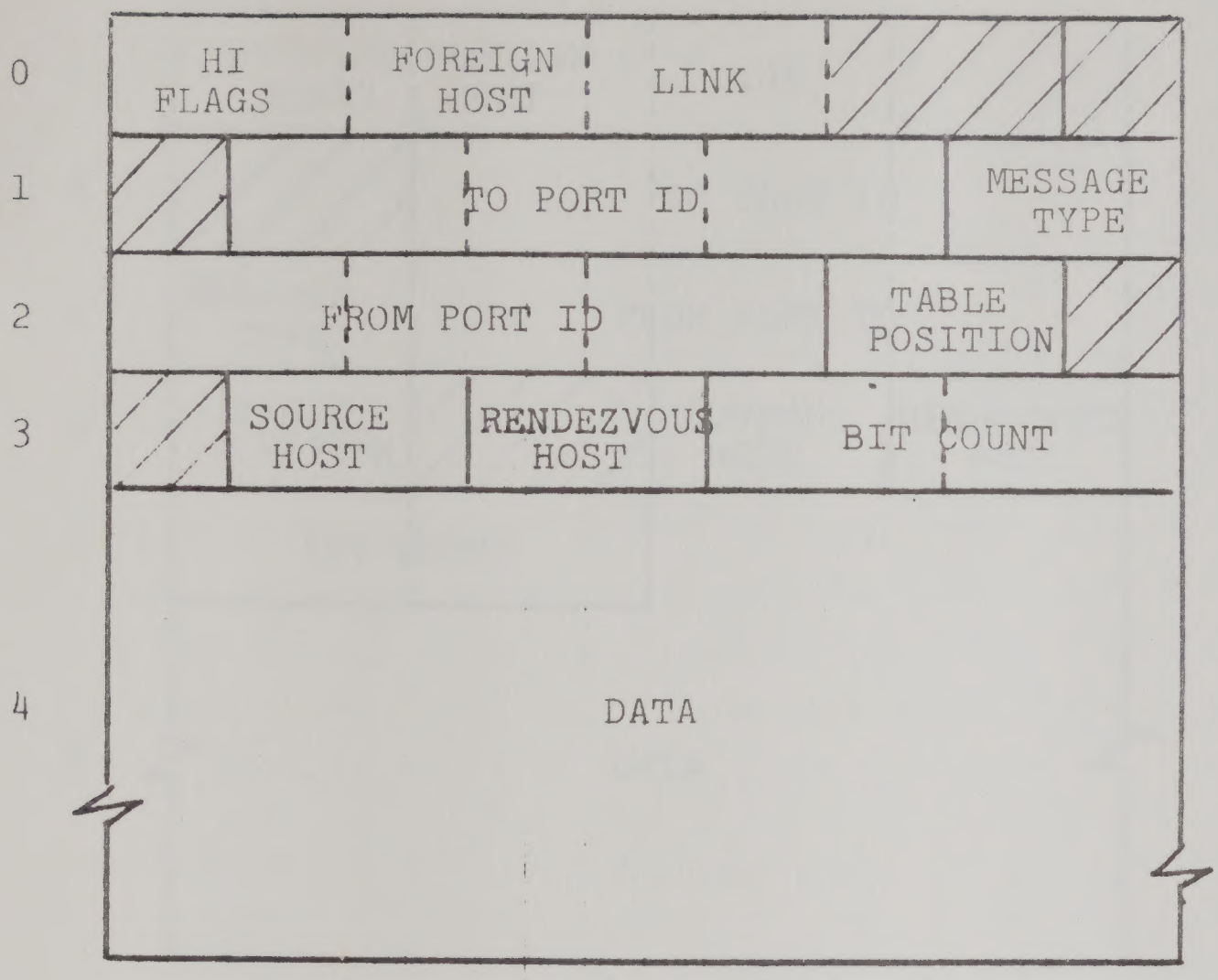



16-bit Host format



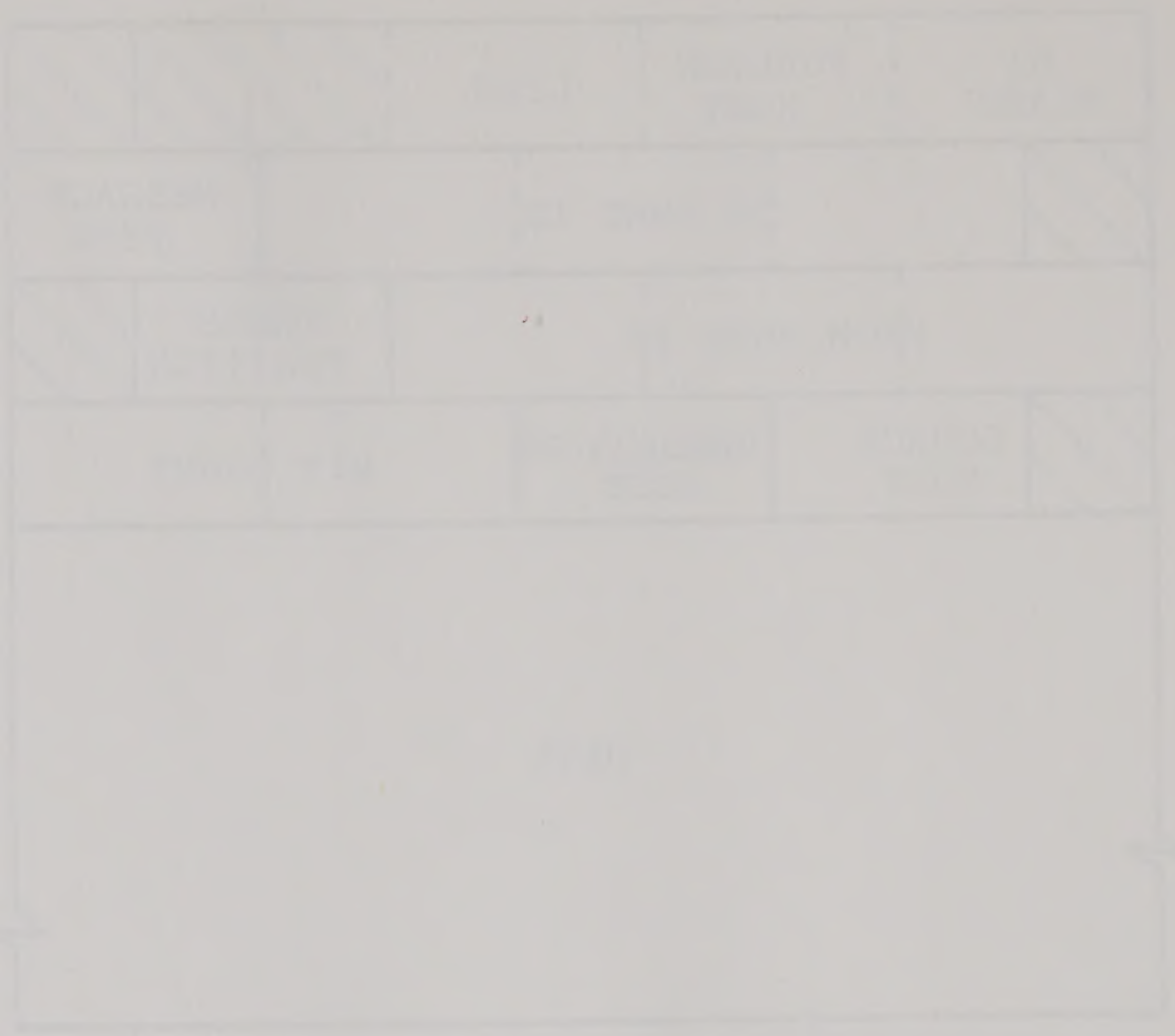



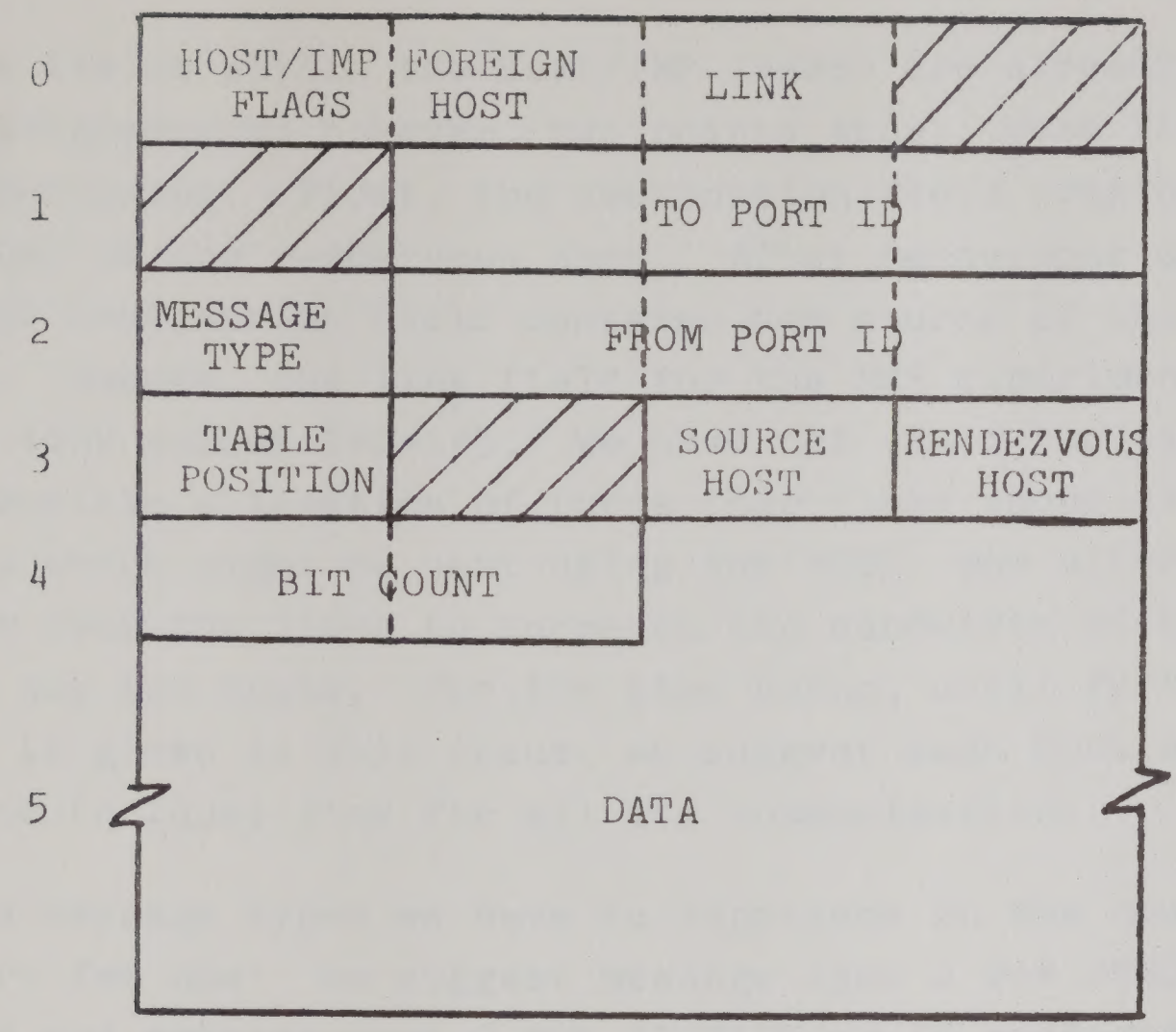
= unused



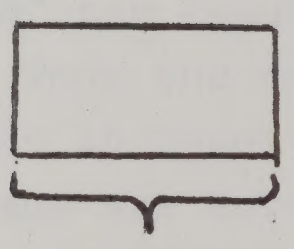

36-bit Host format

8 bits

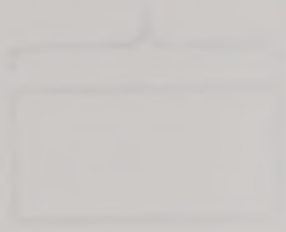





32-bit Host format



8 bits



The fields within the Host/IMP leader are already familiar to NCP programmers; however, two points about these fields are worth mentioning. First, the destination field originally contains the number of the rendezvous Host. After rendezvous at an intermediate site, the destination field contains the source of the message rendezvoed with. Second, the link field for the MSP experiment can only contain link number 192-195. We have not taken the time to figure out a sensible allocation of these four links among all the messages which might be sent using the MSP. One alternative is to cycle over the links to increase the bandwidth of the "pipe" between any two Hosts. For the time being, until further consideration is given to this issue, we suggest each Host at a site using one (unique) link for all its communication.

The message types we have to represent in the message type field are few now: we suggest message type 2 for SEND or OUT messages and message type 3 for RECEIVE or IN messages. Message type 4 is the FLUSH message, if FLUSH is used.

The rendezvous Host field needs no comment. Except that the field is unnecessary after the rendezvous has taken place and could then be used for something else.

The bit count is a count of the data bits in an OUT message or the size of the input buffer (not including the header) in an IN message. Thus the sender process can tell from the IN message bit count when it receives the IN message how much of the data in the OUT message was accepted by the receiver process and can use this knowledge to retransmit the remainder of the message if so desired. After the rendezvous, we recommend that all of the data in the message be sent on the source of the IN message even if the OUT bit count was greater than the IN bit count. Thus, at the receiver Host the monitor has the option (if it wants to take it) of discarding the message for being too long, sending the number of bits the receiver process has done an in for into the receiver process and discarding the rest, or queueing the rest of the bits and somehow notify the receiver process that there are more bits which the receiver process can ask for.

The to- and from-port-id fields are 24-bit numbers. This size was chosen to help the TIPs. The first eight bits of a port id should be the number of the Host at which this port id was created. Note well, that this is not necessarily the Host at which the port is being used. This is necessary since rendezvous take place at intermediate sites and because ports may move from site to site. We suggest that all port ids with the first eight bits all zero be reserved for network-wide use. In particular, a port id with all 24 bits zero will be used to mean "ANY". This gives us the options of:

RECEIVE from ANY to SPECIFIC
RECEIVE from SPECIFIC to SPECIFIC
SEND from SPECIFIC to ANY
and SEND from SPECIFIC to SPECIFIC

Examples of the use of these options will be given below.

The other options (RECEIVE to ANY) and (SEND from ANY) we feel are kind of useless but would not prohibit them. We believe that in the absence of explicit specification of rendezvous Host, the use of an ANY port id in the user's system call should affect the default rendezvous site as follows:

RECEIVE from ANY -- rendezvous in receiver
RECEIVE from SPECIFIC -- rendezvous in sender
SEND to ANY -- rendezvous in sender
SEND to SPECIFIC -- rendezvous in sender

The less significant 16 bits of the port id can be used however a Host wants to. For instance, eight bits might be used as a process id and eight bits might be used as a channel specification within the specified process. We suggest that each Host reserve the port ids with the middle eight bits all zero for special uses as well known ports.

The table position field is included to help prevent costly table searches at interrupt level. Hosts sending INs and OUTs, put in the table position field the rendezvous table position of the SEND or

RECEIVE associated with the IN or OUT. At an intermediate Host rendezvous, the table position fields in the matching IN and OUT are swapped so that when the messages arrive at the opposite end, the matching SEND and RECEIVE can be found quickly. The MSP must do the swap at the rendezvous, but of course the MSPs need not fill in the table position field when first transmitting an IN or OUT in which case the information arriving in an IN or OUT will be meaningless. The general algorithm, then, is to check the table position as specified in this field and if that fails, search the whole table.

The source field is filled in in INs and OUTs by the MSP which originally sends these messages. At the rendezvous the source of each message is preserved in the message being forwarded to the final Host. When an IN or OUT arrives at a process, the process can use the source information to update its understanding of the rendezvous Host (e.g., when the destination Host and rendezvous Host are different).

Examples

The typical example.

We envision commmunication normally taking place using specific to and from ports and rendezvous at the sender. For instance, the TIP would probably send to other Hosts using this method and would certainly receive from other Hosts using this method, since no data can come from the Host until the TIP asks for it. In this "normal" method a monitor could even look at the bit count in the arriving IN-message, use that as an allocation and then simulate an OUT-message of the exact correct length.

The logging example.

Consider an example of SEND to SPECIFIC and RECEIVE from ANY with the rendezvous at the receiver. This method might be used by some logging receiver process with a well-known to-port. For instance, a measurements program to which statistics are sent from many processes throughout the net.

The program library example.

Suppose within a given time-sharing system there is a particular library routine which is available for use by any process in the network. The library process has a RECEIVE from ANY always pending at a well-known port. Eventually, some process sends a message to the library process' well-known port. This message includes the data to be processed, a port to use for sending the answer, and some money. The library process takes some of the money and sends it to the well-known port of the accounting process which itself has a RECEIVE from ANY pending. The library process then processes the data and sends the answer back to the process which requested the service using a SEND to SPECIFIC message which rendezvous at the destination where there is already a RECEIVE from SPECIFIC pending. Of course, in this message besides the answer, any change the requesting process has coming is returned.

A comment.

As can be seen from our examples, we think rendezvousing at an intermediate Host will seldom be done as the chief benefit of this comes when it is desirable to move a port (see reference 4 for a discussion of this). We would like to see all Hosts provide some (meager) amount of buffering for this purpose but would not require it. It shouldn't be too painful to provide a little of this kind of buffering--especially since a Host can throw away any message it can't handle.

(THIS PAGE WILL BE REPLACED WITH A BETTER DESCRIPTION
OF TELNET UNDER MSP
IN A FEW DAYS -- DCW)

TELNET

Let us postulate a pair of Telnet programs that maintain two bidirectional communication paths, one for data and one for control. Let us also assume, for convenience that the port IDs are as follows:

If the WRITE-CONTROL-ID is N, then--
READ-CONTROL-ID = N+1,
WRITE-DATA = N+2,
READ-DATA = N+3.

The initial state is the server Telnet sitting with a READ-FROM-ANY pending.

The user Telnet now issues a SEND-TO-SPECIFIC with the data field containing the PORT-ID of the SERVER's WRITE-CONTROL-ID. This message is sent from the user-Telnet's WRITE-CONTROL-ID.

Thus all port IDs are specified by the user Telnet, so, if desired, he need only remember one number and derive the rest. Uniqueness is preserved since the port IDs supplied by the user Telnet contain his Host ID and other information making the ID unique to him.

Now that these communication paths are established, the two processes can exchange data and control information according to established Telnet protocols.

## The Information Operator

The Message Switching Protocol itself imposes no fixed requirements on the use of the port ID's, and the problem of process identification is somewhat separated from the means used to effect communication. It is, however, very much a part of the overall issue of interprocess communication, and so we here specify a facility for handling process identification, the information operator.

One goal in a process identification scheme is to provide a means by which processes can select their own identifiers which can be guaranteed unique and can contain information meaningful to the user. Problems of efficiency prevent making the port ID's themselves large enough to accomplish this aim. Efficiency questions aside, it would appear to be ideal to allow processes to use character strings of arbitrary length to identify themselves. Uniqueness can then be easily ensured if, for example, users follow the convention of including their names in the process identification string. Further, the remainder of the name can be chosen to have some meaning related to its use with obvious advantages and convenience for users.

One solution is to establish a convention whereby the symbolic identifiers are used only during some initial phase of communication and not in every message. That is, processes identify each other initially using symbolic identifiers, but exchange local port identifiers at that time which are used for all ensuing messages.

The means of providing this facility is to establish a process at each of a number of Hosts (e.g., all server Hosts) called the "information operator". The function of this process is to associate symbolic identification strings and port ID's. A process can identify itself and/or a foreign process to the information

operator, and may request the port ID of the foreign process. The symbolic identification strings are chosen by the processes and are long enough to contain meaningful information, e.g., LOGGER, MURPHY-TESTPROG.

Communication with the information operator, whether by local or remote processes, is via the regular MSP functions. The information operator will always have a RECEIVE ANY outstanding on a well-known port. This could in general be the only well-known port in existence. A message received on this port contains the following parameters:

1. String identifying the foreign process with which communication is desired.

2. String identifying the calling process.

3. Calling process' port number.

4. A delay specification.

The format of these parameters is shown in Fig. 4. In some cases, one or more of the arguments would be null. Following receipt of a message, the information operator will, in some cases, do a SEND SPECIFIC to the calling process' port number providing the desired information or notice of failure.

The following two cases would appear to cover all functions of the information operator. They correspond to the SEND/RECEIVE SPECIFIC and SEND/RECEIVE ANY cases of the MSP.

1. Two processes each knowing the specific identity of the other wish to communicate. Each does a SEND SPECIFIC to the information operator, giving parameters 1-3, the default delay spec in this case being WAIT. When the information operator receives the second of these and notes that a match exists, it sends to each process the port ID

of the other process and deletes both strings and both port ID's from its tables. The two processes, which have each done a RECEIVE SPECIFIC in anticipation of the foreign port number, can then communicate using just the port numbers and basic MSP functions.

2. A process is set up to provide some sort of general service or information, and its name and protocol advertised. This process intends to maintain an outstanding SEND or RECEIVE ANY for the first (and perhaps only) message transaction, e.g., the library process discussed earlier. Most such processes would be receivers initially, but there might be a few cases where a SEND could be left outstanding, and a foreign process could come along and pick up the information. In either case, the service process will do SEND SPECIFIC to the information operator giving the local symbolic ID and local port ID. The foreign symbolic ID would be null, and the default delay spec is NO-WAIT. That is,

INFO (-, local ID, local port)

The information operator will enter this information in its tables but return nothing to the caller. The caller would proceed to do its SEND/RECEIVE ANY to wait for business. When another process wishes to use the advertised service, it asks the logger for the port ID of the service process, i.e.,

INFO (service ID, -, local port)

The local symbolic ID need not be specified, and the default delay spec is NO-WAIT. The information operator would SEND the port ID of the service process to the

local port of the caller, and retain the table entry for future callers. Only the service process could request the entry be deleted. If the service ID was unknown to the information operator at the time of this call, it would immediately return a failure indication, i.e., Ø.

Communicating processes would normally use the information operator local to one or the other, and like the rendezvous Host in the MSP, this would be agreed upon in advance. Service processes would normally use the information operator at their local site, and correspondingly, user processes would call the information operator at the site where the service process was expected to be available. There is no restriction on using an information operator at some other site of course, and some small and/or lazy servers could use a different Host for their service process ID's. It presents no problem for two or more information operators to have entries for the same service process, and in fact, this may be very desirable for special types of service processes which exist only one place on the net and may move around from time to time.

Processes would specify their own local port numbers, and each system would have to provide some way to help user processes do this. In TENEX for example, one would probably use the job number concatenated with another number assigned within the job. The information operator cannot supply port numbers because it will be running on a different Host than one or both of the communicants and cannot know what is a unique number for that Host. In some cases, processes would ask the "unique number process" (described below) for their local port ID, and would make it known via the information operator.

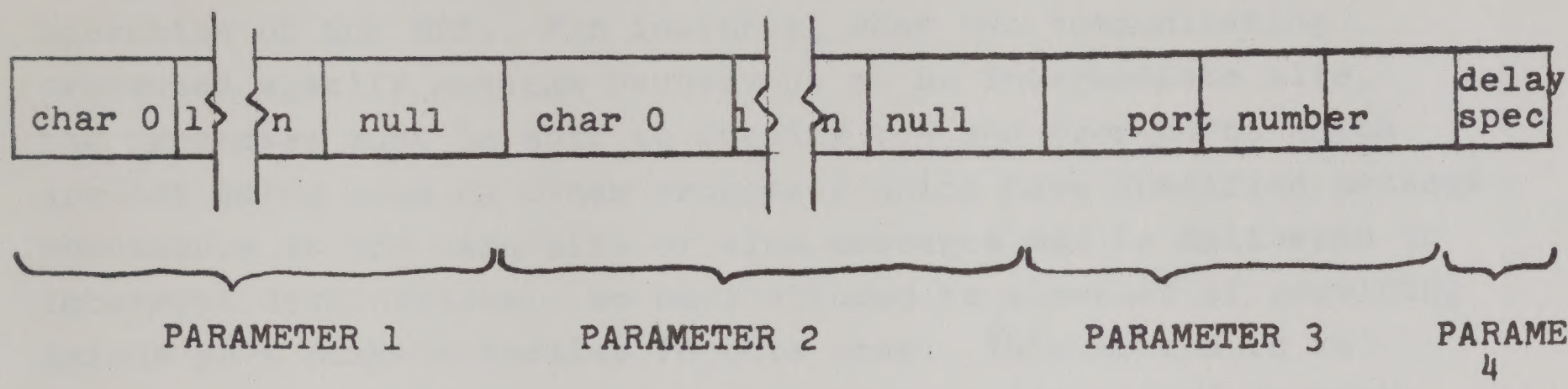
In actual practice, a few exceptions would be made to the rule that the only "well-known" port in the world is the information operator. Such exceptions would be processes common to many Hosts, e.g., LOGGER, or those in particularly frequent use. In such cases

the unique port numbers would be assigned by administrative fiat and recorded and published to all users.

The symbolic identification strings are specified to be from 1 to 39 (an arbitrary maximum) ASCII characters terminated by a null (byte of all Ø's). The characters will be 7-bit ASCII in 8-bit bytes with the high order bit set to Ø. A null string (first byte is null) is used where no argument is required.

Format of Information Operator Messages

To Information Operator: A stream of 8-bit bytes.




Parameters given:

1. String identifying the foreign process with which communication is desired. (1 to 39 characters, or null)

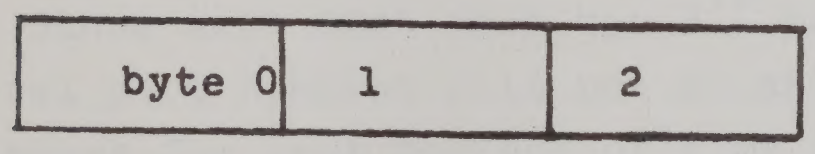
2. String identifying the calling process. (1 to 39 characters, or null)

3. Calling process' port number.

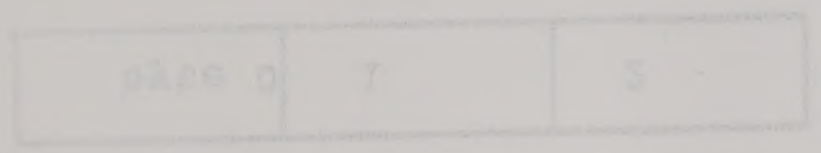
4. Delay specification:
   0 = default
   1 = wait for match
   2 = don't wait for match

---

From Information Operator: 3 8-bit bytes.




Port number (24 bits) of requested foreign port if successful, 0 if unsuccessful.



Unique Port Numbers

The existence of unique port numbers is essential to the operation of the MSP. For instance, when two communicating processes specify message rendezvous at an intermediate site, the processes must be able to specify to- and from-ports which are not being used by other processes which have specified message rendezvous at the same site or else messages may be delivered to incorrect destinations. We have alluded to a method of providing unique port numbers earlier in this note. This method is to partition the 24-bit port number space into disjointed segments and give one segment to each Host in the network to distribute when it is called upon to "create" a unique port id. Thus each 24-bit Host number will consist of two major parts. The first 8 bits will be the number of the Host "creating" the port id and the next 16 bits can be used in any manner the creating Host desires. This gives each Host $2^{16}$ port numbers to distribute, and each Host will have the burden of distributing its segment of the port number space in a unique manner. We recommend the convention that the port numbers with the middle 8 bits equal to zero be reserved for well-known ports in the creating Host's system. We already recommended in an earlier section that port numbers with the first 8 bits equal to zero be reserved for network-wide use and in particular the port number with all 24 bits equal to zero be used to mean ANY.

Since each Host only has $2^{16}$ port numbers to distribute, in general port numbers will not be able to be held and used by processes for long periods of time (e.g., weeks and months). More typically, Hosts will probably implicitly "take back" all port numbers the Host has distributed each time the Host's system goes down and will redistribute the port numbers as required when the system comes back up. In other words, port numbers will not in general remain unique over the going down of the creating Hosts.

Of course, a given Host may see fit to give the same port numbers to a number of standard processes (such as the FORTRAN compiler) each time it comes up; port numbers registered with an information operator will frequently remain constant over system ups and downs.

In spite of the fact that each Host will probably not in general be able to distribute port numbers to arbitrary user processes which can be guaranteed to remain unique over a long period of time, there will still be demand for provision of long-term unique port numbers. To some, the procedure of going through the information operator smacks much too much of making a connection. These people will insist that for a variety of reasons their processes be allowed to communicate via ports whose identifiers remain constant for long periods of time. Therefore, it would be nice if at one or two places in the network, a long-term unique number service was provided. We'll call a process providing this service the Unique Number Process. The Unique Number Process would have assigned to it one segment of the unique port number space—all those port numbers, for instance, with the first 8-bits equal to $377_8$. This process would have a SEND-to-ANY pending from a well-known port with local rendezvous specified. When any process wanted a unique number which it could depend on not to be used for all time or until the number is given back, it would send a RECEIVE-from-SPECIFIC specifying the well-known port of the Unique Number Process and rendezvous at the Unique Number Process' Host. The Unique Number Process' pending SEND-to-ANY would contain a unique number. Also, the Unique Number Process would have a RECEIVE-from-ANY always pending at another well-known port with local rendezvous specified. At this port the Unique Number Process would receive unique numbers which processes are giving back. The Unique Number Process would maintain a bit table $2^{16}$ bits long indicating the state of each of its unique numbers (free or in use) in some long-term storage medium such as in the file system. The Unique Number Process might also maintain some information about each process to which it gives a unique number

so that when the supply of unique number gets depleted, processes can be asked to return them.

It has already been mentioned that some of the process ID's registered along with their symbolic names at the information operator might be long-term unique numbers gotten from the Unique Number Process. It should also be mentioned that there would seem to be no reason, other than scarcity of storage space, that in addition to the port number through which primary access is gained to a process and which was called the process ID in the previous section, arbitrary port numbers along with their symbolic identifiers could not be registered with an information operator. For instance, rather than registering the name BBN-FORTRAN and a single port number, one could perhaps register the port numbers whose symbolic identifiers were BBN-FORTRAN-CONTROL-TELETYPE, BBN-FORTRAN-INPUT-FILE, BBN-FORTRAN-LISTING-FILE, and BBN-FORTRAN-BINARY-OUTPUT-FILE. This is perhaps at odds with standard practice within operating systems, but is consistent with the philosophy of reference 4 that communication is done with ports and not processes.

Let us now address an issue which has been ignored up to now and which was only alluded to in reference 4, the issue of port protection. We have not given this matter a great deal of thought; however, one mechanism for port protection seems quite straightforward. The heart of this mechanism is a process at each Host which we shall call (alliteratively) the Port Protection Process (PPP). The PPP maintains a list of all processes which exist at the Host and for each process the numbers of all ports which the process has "legally" obtained. Every time a process does a SEND or RECEIVE, the monitor checks with the PPP to see if the process has specified port numbers it has the right to use; i.e., those legally obtained. The PPP has some RECEIVEs always pending at well-known ports. When one process wants to pass a port to some other process, the first process sends a message

to the PPP specifying the number of the port to be sent, the Host number at which the second process resides, a port at which the second process is expecting to receive the port, etc. The PPP looks up in its tables whether the first process has the port it wants to send. If it does, it sends a message to the PPP at the destination site. The message contains the number of the port to be transferred and the RECEIVE port for the destination process. The destination PPP checks in its table whether the process has the RECEIVE port, and if so, passes the new port to the process and updates its tables to indicate the process now possesses the new port. The messages to a PPP will optionally be able to specify that a copy of a port be sent, a port be deleted, etc. The PPPs would probably have some built-in legal ports for each process, particularly the port's processes used to communicate with the PPP. The exact specification requires development but that should not be hard (see (3), (6), and (7) in reference 4). The main difficulty we see is efficient checking of the PPP's tables by the monitor for every RECEIVE or SEND without entirely supplanting the monitor's current protection system.

Flow Chart

The following section describes a flow chart for most of the MSP. A distinction is made between calls made by local processes called SEND and RECEIVE, and messages coming in over the NET called IN and OUT. An additional distinction is made between calls (or messages) with a local rendezvous and those with a foreign rendezvous Host.

Since the code is quite similar, the distinction need not be made, but will be included for the sake of clarity.

It is assumed that the MSP has table provisions for the following items:

source of message
rendezvous Host
FROM-PORT-ID
TO-PORT-ID
table position
type of message
data size and location
data about the user process

User does a SEND or RECEIVE

A. Rendezvous is at a foreign host

    1. Store the appropriate table data

    2. send a message to the rendezvous host

        a. SEND: OUT + DATA

        b. RECEIVE: IN

B. Rendezvous is local - look for entry in table

    1. Entry NOT found: create entry with appropriate data

    2. A matching entry exists in the table:

        a. RECEIVE: give user the data

        b. Send a message to the other host (as specified by the source field of the original msg)

            1) SEND: OUT + DATA

            2) RECEIVE: IN

        c. Alert user to the fact that transaction is complete

        d. Clear table entry

An IN is received over the NET - search table for matching entry.

A. No matching entry  create an entry with appropriate data.

B. A match exists

1. Entry was cause by a local SEND

a. Send "OUT _ DATA" to source of IN

b. Inform user of transaction

c. Clear table entry

C 2. Entry was caused by an OUT received over net - acting as third host.

a. Send IN to site that created table entry

b. Send OUT + DATA (previously buffered) to site sending the IN

c. Clear table entry.

An OUT + DATA is received over the NET - search table for matching entry

A. No match is found

    1. buffer data

    2. create appropriate table information

B. A match is found

    1. Table entry was caused by locally executed RECEIVE

        a. give data to the user and alert him to its existence

        b. send a matching "IN" to the source of the "OUT"

        c. remove entry from table.

    2. Table entry was caused by the receipt of an "IN" over the NET, thus we are acting as a third party host

        a. send the "OUT + DATA" to the host stored in the table

        b. send an "IN" to the host from which the "OUT" had just arrived.

MSP Variations

It may be of interest to the reader to know of some of the other MSPs we have considered while arriving at the present one.

The simplest we considered is an MSP based on all rendezvous being done at the destination Host. The sender process sends an OUT-message plus the data to the destination Host. The receiver process does an IN which stays at the receiver's Host. The OUT and RECEIVE rendezvous and the data is passed to the receiver process. The transmission is now complete, except in some variations of this MSP an acknowledgment is sent to the sender process. This MSP has a couple of disadvantages: In the simplest formulation, the RECEIVE had to be waiting when the OUT+data arrived, otherwise the out and data were thrown away. This puts too tight a constraint on the timing of the SEND and RECEIVE, especially since the sender and receiver processes can be a continent apart. However, if the IN is allowed to arrive first and must be held until matched by a RECEIVE, the monitor must buffer an indeterminate amount of data in all cases including the normal one. Further, basing everything on rendezvous at the destination makes the process of moving a port difficult.

The next simplest MSP we considered was the IPC of reference 4. This works just the opposite of the above described MSP in that it is based on almost all rendezvous being done at the source Host with two special messages to handle the relatively uncommon cases when a rendezvous must be done at the destination or an intermediate Host. This system, its advantages, and disadvantages is discussed at very great length in the reference.

A third variation on the MSP, suggested by Crowther, is the same as the present MSP in that the OUT and IN rendezvous at a process specified rendezvous Host and the OUT is sent to the source of the IN and the IN to the source of the OUT, but the data is not sent along with the OUT. Instead, when the OUT finally reaches the

source of the IN, another message is sent from the receiver Host to the source Host requesting the data to be sent. The data finally is transmitted to the destination in response to this data request message. Our main objection to this system is its lack of symmetry, but we do recognize that it does not require any Host to buffer data for which a process has not set up an input buffer and perhaps for that reason it is a better system than the MSP we are presenting.

In the last MSP variation we considered, the difference between SEND or RECEIVE and OUT or IN was discarded. In this case only one message is used which we will call TRANSFER. When a process executes a TRANSFER it can specify an input buffer, an output buffer, both, or neither. Two processes wishing to communicate both execute TRANSFERs specifying the same to and from port ids and the same rendezvous Host. The TRANSFERs result in TRANSFER-messages plus data in the case that an output buffer was specified which rendezvous at the rendezvous Host. When the rendezvous occurs, the TRANSFER-messages plus their data cross and each is sent to the source of the other. The system allows processes not to know whether they must do a SEND, or RECEIVE and is (perhaps) a nice generalization of the MSP presented in this note. For instance, two processes can exchange data using this system, or two processes can kind of interrupt each other by sending dataless TRANSFERs. This variation of the MSP is a development of a suggestion of Steve Crocker. Its disadvantages are: (1) unintentional matches are more likely to occur, (2) rendezvous selection site is more complex, and (3) it's hard to think about.

Appendix

A System for Interprocess Communication in a Resource Sharing Computer Network. Communications of the ACM, April, 1972. Permission to reprint this paper was granted by permission of the Association for Computing Machinery.

N.B. The ideas of section 4 of the following paper are in no way critical to the ideas developed in section 3--DCW.

# A System for Interprocess Communication in a Resource Sharing Computer Network

David C. Walden
Bolt Beranek and Newman Inc.*

**A system of communication between processes in a time-sharing system is described and the communication system is extended so that it may be used between processes distributed throughout a computer network. The hypothetical application of the system to an existing network is discussed.**

**Key Words and Phrases: interprocess communication, time-sharing, computer networks, resource sharing**
**CR Categories: 3.81, 4.39, 4.82, 4.9**

## 1. Introduction

A resource sharing computer network is defined to be a set of autonomous, independent computer systems, interconnected to permit each computer system to utilize all the resources of the other computer systems much as it would normally call one of its own subroutines. This definition of a network and the desirability of such a network are expounded upon by Roberts and Wessler in [9]. Examples of resource sharing could include a program filing some data in the file system of another computer system, two programs in remote computer systems exchanging communications, or users simply utilizing programs of another computer system via their own.

The actual act of resource sharing can be performed in two ways: in an ad hoc manner between all pairs of computer systems in the network; or according to a systematic network-wide standard. This paper develops a possible network-wide system for resource sharing.

It is useful to think of resources as being associated with processes[1] and available only through communication with these processes. This is a viewpoint which has been successfully applied to time-sharing systems [1] and has more recently been suggested to be an appropriate view for computer networks [8]. Consistent with this view, the fundamental problem of resource sharing is held in this paper to be the problem of inter-

[1] Almost any of the common definitions of a process would suit the needs of this paper.

Copyright © 1972, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

* 50 Moulton Street, Cambridge, MA 02138. The work was primarily supported by the Advanced Research Projects Agency under Contract DAHC 15-69-C-0179.

process communication. With Carr, Crocker, and Cerf [2], the view is also held that interprocess communication over a network is a subcase of general interprocess communication in a multiprogrammed environment.

These views have led to a two-part study. First, a set of operations enabling interprocess communication within a single time-sharing system is constructed. This set of operations eschews many of the interprocess communication techniques currently in use within time-sharing systems—such as communication through shared memory—and relies instead on techniques that can be easily generalized to permit communication between remote processes. The second part of the study presents such a generalization. The hypothetical application of this generalized system to the ARPA Computer Network [9] is also discussed.

This paper does not represent a Bolt Beranek and Newman position on Host protocol for the ARPA network.

## 2. A System for Interprocess Communication Within a Time-Sharing System

This section describes a set of operations enabling interprocess communication within a time-sharing system. Following the notation of [10], this interprocess communication facility is called an IPC. As an aid to the presentation of this IPC, a model for a time-sharing system is described; the model is then used to illustrate the use of the interprocess communication operations.

The model time-sharing system has two pieces: the monitor and the processes. The monitor performs such functions as switching control from one process to another process when a process has used "enough" time, fielding hardware interrupts, managing core and the swapping medium, controlling the passing of control from one process to another (i.e. protection mechanisms), creating processes, caring for sleeping processes, and providing to the processes a set of machine extending operations (often called "supervisor" or "monitor" calls).

The processes perform the normal user functions (user processes) as well as the functions usually thought of as being supervisor functions in a time-sharing system (system processes) but not performed by the monitor in the current model. A typical system process is the disk handler or the file system. System processes are probably allowed to execute in supervisor mode, and they actually execute I/O instructions and perform other privileged operations that user processes are not allowed to perform. In all other ways, user and system processes are identical. For reasons of efficiency, it may be useful to think of system processes as being locked in core.

Although they will be of concern later in this study, protection considerations are not our concern here; instead, all processes are assumed to be "good" processes which never make any mistakes. If the reader needs a protection structure to keep in mind while he reads this paper, the *capability* system developed in [1, 3, 6, and 7] should be satisfying.

Of the operations a process can call on the monitor to perform, the five following are of particular interest for providing a capability for interprocess communication.

RECEIVE. This operation allows a specified process to send a message to the process executing the RECEIVE. The operation has four parameters: the port (defined below) awaiting the message—the RECEIVE port; the port a message will be accepted from—the SEND port; a specification of the buffer available to receive the message; and a location to transfer to when the transmission is complete—the restart location.

SEND. This operation sends a message from the process executing the SEND to a specified process. It has four parameters: a port to send the message to—the RECEIVE port; the port the message is being sent from—the SEND port; a specification of the buffer containing the message to be sent; and the restart location.

RECEIVE ANY. This operation allows *any* process to send a message to the process executing the RECEIVE ANY. The operation has four parameters: the port awaiting the message—the RECEIVE port; a specification of the buffer available to receive the message; a restart location; and a location where the port which sent the message may be noted.

SLEEP. This operation allows the currently running process to put itself to sleep pending the completion of an event. The operation has one optional parameter—an event to be waited for. An example event is the arrival of a hardware interrupt. The monitor never unilaterally puts a process to sleep as a result of the process executing one of the above four operations; however, if a process is asleep when one of the above four operations is satisfied, the process is awakened.

UNIQUE. This operation obtains a unique number from the monitor.

A *port* is a particular data path to a process (a RECEIVE port) or from a process (a SEND port), and all ports have an associated unique *port number* which is used to identify the port. Ports are used in transmitting messages from one process to another in the following manner. Consider two processes, $A$ and $B$, that wish to communicate. Process $A$ executes a RECEIVE to port $N$ from port $M$. Process $B$ executes a SEND to port $N$ from port $M$. The monitor matches up the port numbers and transfers the message from process $B$ to process $A$. As soon as the buffer has been fully transmitted out of process $B$, process $B$ is restarted at the location specified in the SEND operation. As soon as the message is fully received at process $A$, process $A$ is restarted at the location specified in the RECEIVE operation.[3] How the processes come by the correct port numbers with which to communicate with other processes is discussed later.

Somewhere in the monitor there must be a table of port numbers associated with processes and restart locations. *The table entries are cleared after each SEND/*

*RECEIVE match is made.* When a SEND is executed, nothing happens until a matching RECEIVE is executed. If a proper RECEIVE is not executed for some time, the SEND is timed out after a while and the SENDing process is notified. If a RECEIVE is executed but the matching SEND does not happen for a long time, the RECEIVE is timed out and the RECEIVing process is notified. A RECEIVE ANY never times out, but may be taken back using a supervisor call.

The mechanism of timing out "unused" table entries is of little fundamental importance—it merely provides a convenient method of garbage-collecting the table. There is no problem if an entry is timed out prematurely, because the process can always reexecute the operation. However, the timeout interval should be long enough so that continual reexecution of an operation will cause little overhead. If the table where the SEND and RECEIVE are matched up ever overflows, a process originating a further SEND or RECEIVE is notified just as if the SEND or RECEIVE timed out.

The *restart location* is an interrupt entrance associated with a pseudo interrupt local to the process executing the operation specifying the restart location. If the process is running when the event causing the pseudo interrupt occurs (for example, a message arrives satisfying a pending RECEIVE), the effect is exactly as if the hardware interrupted the process and transferred control to the restart location. Enough information is saved for the process to continue execution at the point it was interrupted after the interrupt is serviced. If the process is asleep, it is readied and the pseudo interrupt is saved until the process runs again, and the interrupt is then allowed. Any RECEIVE or RECEIVE ANY message port may thus be used to provide process interrupts, event channels, process synchronization, message transfers, etc. The user programs what he wants.

Most time-sharing systems could be made to provide the five operations described above with relative ease, since these are only additional supervisor calls.

*Example.* Suppose that our model time-sharing system is initialized to have several processes always running. Additionally, these permanent processes have some universally known and permanently assigned ports.[3] Suppose that two of the permanently running processes are the logger-process and the teletype-scanner-process. When the teletype-scanner-process first starts running, it puts itself to sleep awaiting an interrupt from the hardware teletype scanner. The logger-process initially puts itself to sleep awaiting a message from the teletype-scanner-process via well-known permanent SEND and RECEIVE ports. The teletype-scanner-process keeps a table indexed by teletype number, containing in each entry a pair of port numbers to use to send characters from that teletype to a process and a pair of port numbers to use to receive characters for that teletype from a process. If a character arrives (waking up the teletype-scanner-process) and the process does not have any entry for that teletype, it gets a pair of unique numbers from the monitor (via UNIQUE) and sends a message containing this pair of numbers to the logger-process, using the ports for which the logger-process is known to have a RECEIVE pending. The scanner-process also enters the pair of numbers in the teletype table, and sends the character and all future characters from this teletype to the port with the first number from the port with the second number. The scanner-process must also pass a second pair of unique numbers to the logger-process for it to use for teletype output and do a RECEIVE using these port numbers. When the logger-process receives the message from the scanner-process, it may, for instance, start up a copy of what SDS 940 TSS [5] users call the executive[4] and pass the port numbers to this copy of the executive, so that this executive-process can also do its inputs and outputs to the teletype using these ports. If the logger-process wants to get a job number and password from the user, it can temporarily use the port numbers to communicate with the user before it passes them on to the executive. The scanner-process could always use the same port numbers for a particular teletype as long as the numbers were passed on to only one copy of the executive at a time.

It is important to distinguish between the act of passing a port from one process to another and the act of passing a port *number* from one process to another. In the previous example, where characters from a particular teletype are sent either to the logger-process or an executive-process by the teletype-scanner-process, the SEND port always remains in the teletype-scanner-process, while the RECEIVE port moves from the logger-process to the executive-process. On the other hand, the SEND port number is passed between the logger-process and the executive-process to enable the RECEIVE process to do a RECEIVE from the correct SEND port. It is crucial that, once a process transfers a *port* to some other process, the first process no longer uses the port. A mechanism that enforces this, such as the protected object system of [7], could be added. Using this mechanism, a process executing a SEND would need a capability for the SEND port, and only one capability for this SEND port would exist in the system at any given time. Likewise a process executing a RECEIVE would be required to have a capability for the RECEIVE port, and only one capability for this RECEIVE port would exist at a given time. Without such a protection mechanism, a port implicitly moves from one process to another by the processes merely using the port at disjoint times, even if the port's number is never explicitly passed.

[3] Interestingly, there seems no reason to prevent (and some reasons for allowing) a process concurrently having several outstanding RECEIVEs from different ports to a given port. For instance, process *A* might execute a RECEIVE to port *N* from port *R* concurrently with the above RECEIVE to port *N* from port *M*.

[3] Or perhaps there is only one permanently known port, which belongs to a directory-process that keeps a table of permanent-process/well-known-port associations.

[4] That program which prints file directories, tells who is on other teletypes, runs subsystems, etc.

Of course, if the protected object system is available to us, there is really no need for two port numbers to be specified before a transmission can take place. The fact that a process knows an existing RECEIVE port number could be considered *prima facie* evidence of the process's right to send to that port. The difference between RECEIVE and RECEIVE ANY ports then depends solely on the number of copies of a particular port number that have been passed out. A system based on this approach would clearly be preferable to the one described here if it was possible to assume that all autonomous time-sharing systems in a network would adopt this protection mechanism. If this assumption cannot be made, it seems more practical to require both port numbers.

*Note that in the interprocess communication system (IPC) being described here, when two processes wish to communicate, they set up the connection themselves, and they are free to do it in a mutually convenient manner.* For instance, they can exchange port numbers, or one process can pick all the port numbers and instruct the other process which to use, or they can just *know* the correct port numbers to use.[5] However, in a particular implementation of a time-sharing system, the builders of the system might choose to restrict the processes' execution of SENDS and RECEIVES and might forbid arbitrary passing around of ports and port numbers, requiring instead that the monitor be called (or some other special program) to perform these functions. Generally, well-known permanently-assigned ports are used via RECEIVE ANY. The permanent ports will most often be used for starting processes, and consequently, little data will be sent via them. If a process is running (perhaps asleep) and has a RECEIVE ANY pending, then any process knowing the RECEIVE port number can talk to that process without going through loggers. This is obviously essential within a local time-sharing system and seems very useful in a more general network if the ideal of resource sharing is to be reached. For instance, in a resource sharing network, the programs in the subroutine libraries at all sites might have RECEIVE ANYS always pending over permanently assigned ports with well-known port numbers. Thus, to use a particular network resource such as a matrix inversion subroutine at a site with special matrix manipulation hardware, a process running anywhere in the network can send a message to the matrix inversion subroutine containing the matrix to be inverted and the port numbers to be used for returning the results.

Control of data flow is provided in this IPC by the simple method of never starting data transmission resultant from a SEND from one process until a RECEIVE is executed by the receiver. Of course, interprocess messages may also be sent back and forth suggesting that a process stop sending or that space be allocated.

An additional example demonstrates the use of the FORTRAN compiler. We have already explained how a user sits down at his teletype and gets connected to an executive. We go on from there. The user is typing in and out of the executive which is doing SENDS and RECEIVES. Eventually the user types RUN FORTRAN, and the executive asks the monitor to start up a copy of the FORTRAN compiler and passes to FORTRAN as start-up parameters the port numbers the executive was using to talk to the teletype. (Thus, at least conceptually, FORTRAN is passed a port at which to RECEIVE characters from the teletype and a port from which to SEND characters to the teletype.) FORTRAN is, of course, expecting these parameters and does SENDS and RECEIVES via the indicated ports to discover from the user what input and output files the user wants to use. FORTRAN types INPUT FILE? to the user, who responds F001. FORTRAN then sends a message to the file-system-process, which is asleep waiting for something to do. The message is sent via well-known ports, and it asks the file system to open F001 for input. The message also contains a pair of port numbers that the file-system-process can use to send its reply. The file-system looks up F001, opens it for input, makes some entries in its open file tables, and sends back to FORTRAN a message containing the port numbers that FORTRAN can use to read the file. The same procedure is followed for the output file. When the compilation is complete, FORTRAN returns the teletype port numbers (and the ports) back to the executive that has been asleep waiting for a message from FORTRAN and then halts itself. With nothing else to do, the file-system-process goes back to sleep.

## 3. A System for Interprocess Communication Between Remote Processes

The IPC described in the previous section easily generalizes to allow interprocess communication between processes at geographically different locations as, for example, within a computer network.

Consider first a simple configuration of processes distributed around the points of a star. At each point of the star there is an autonomous operating system.[6] A rather large, smart computer system, called the Network Controller, exists at the center of the star. No processes can run in this center system, but rather it should be thought of as an extension of the monitor of each of the operating systems in the network.

If the Network Controller is able to perform the operations SEND, RECEIVE, RECEIVE ANY, and UNIQUE, and if all of the monitors in all of the operating systems in the network do not perform these operations themselves but rather ask the Network Controller to perform

[5] Called ipc-setup in [10].

[6] "Operating system" rather than "time-sharing system" is used in this section to point up the fact that the autonomous systems at the network nodes may be either full-blown time-sharing systems in their own right, an individual process in a larger geographically distributed time-sharing system, or merely autonomous sites wishing to communicate.

these operations for them, then the problem of interprocess communication between remote processes is solved. No further changes are necessary since the Network Controller can keep track of which RECEIVEs have been executed and which SENDs have been executed and match them up just as the monitor did in the model time-sharing system. A network-wide port numbering scheme is also possible with the Network Controller knowing where (i.e. at which site) a particular port is at a particular time.

Next, consider a more complex network in which there is no common center point, making it necessary to distribute the functions performed by the Network Controller among the network nodes. In the rest of this section it will be shown that the set of functions performed by the star Network Controller can be replicated at each of the network sites so that efficient and convenient general interprocess communication between remote processes is still possible. We call all of these network controllers taken together a *distributed Network Controller.*

Some changes must be made to each of the three SEND/RECEIVE operations described above to adapt them for use in a distributed Network Controller. To RECEIVE is added a parameter specifying a site to which the RECEIVE is to be sent. To SEND messages is added a site to which the SEND may be sent, although this is normally the local site. Both RECEIVE and RECEIVE ANY have added the provision for obtaining the source site of any received message. Thus when a RECEIVE is executed, the RECEIVE is sent to the site specified, possibly a remote site. At some (other) time a SEND appears at the same site, normally the local site of the process executing the SEND. (That is, the RECEIVE goes to the SEND site; the SEND does not go anywhere.) At this site, called the *rendezvous site*, the RECEIVE is matched with the proper SEND, and the message transmission is allowed to take place from the SEND site to the site from where the RECEIVE came.

A RECEIVE ANY never leaves its originating site; however, since it must be possible to send a message to a RECEIVE ANY port and not have the message blocked waiting for a RECEIVE at the sending site, we must introduce a new operation which will be called FORCE SEND. The FORCE SEND operation has the same parameters as SEND (a port to send the message to—the RECEIVE port; the port the message is being sent from—the SEND port; a specification of the buffer containing the message to be sent; the restart location; and the site to which to send the message). The message resultant from a FORCE SEND is always sent immediately to the rendezvous site where it is discarded if a proper RECEIVE ANY is not found. An error message is not returned, and acknowledgment, if any, is up to the processes. It is possible to construct a system so that the SEND/RECEIVE rendezvous takes place at the RECEIVE site and eliminates the FORCE SEND operation, but the ability to block a normal SEND transmission at the source site is probably worth the added complexity.

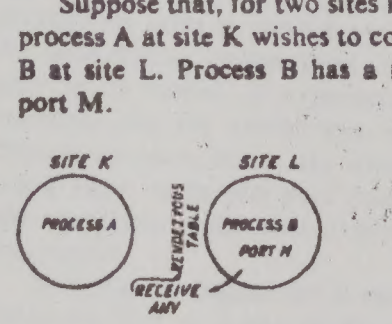
At each site a rendezvous table is kept. This table contains an entry for each unmatched SEND or RECEIVE received at that site and also an entry for all RECEIVE ANYs given at that site. A matching SEND/RECEIVE pair is cleared from the table as soon as the match takes place. As in the similar table kept in the model time-sharing system, SEND and RECEIVE entries are timed out if unmatched for too long, and the originator is notified. RECEIVE ANY entries are cleared from the table when a fulfilling message arrives.

The final change necessary is to give each site a portion of the unique numbers to distribute via its UNIQUE operation. This topic will be discussed further below.

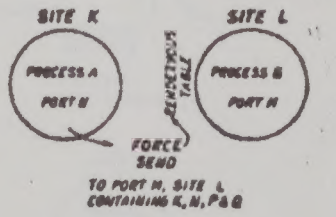
All interprocess communication at a site, to both local and remote processes, should be done via the local portion of the distributed Network Controller. The case is less clear in a star network with a central Network Controller where there is a trade-off between efficiency if local interprocess communication does not have to go through the central Network Controller and increased versatility if it does. The correct solution might be to distribute the Network Controller even for star networks.

To make it clear to the reader how the distributed Network Controller works, an example follows. The details of what process picks port numbers, etc., are only exemplary and are not a standard specified as part of the IPC.

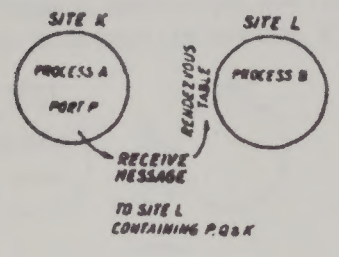
Suppose that, for two sites in the network, K and L, process A at site K wishes to communicate with process B at site L. Process B has a RECEIVE ANY pending at port M.



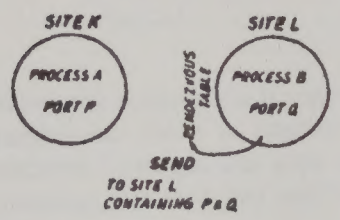

Process A, fortunately, knows of the existence of port M at site L and sends a message using the FORCE SEND operation from port N to port M. The message contains two port numbers and instructions for process B to SEND messages for process A to port P from port Q. Site K's site number is appended to this message along with the message's SEND port, N.



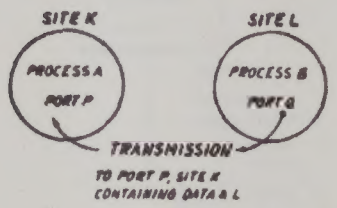
Process A now executes a RECEIVE at port P from port Q. Process A specifies the rendezvous site to be site L.




A RECEIVE message is sent from site K to site L and is entered in the rendezvous table at site L. At some other time, process B executes a SEND to port P from port Q specifying site L as the rendezvous site.

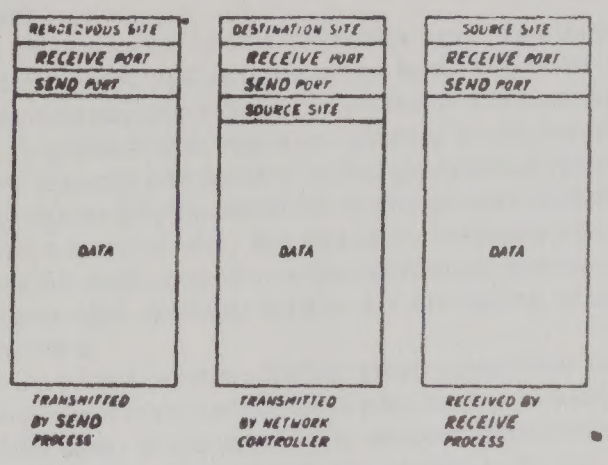



A rendezvous is made, the rendezvous table entry is cleared, and the transmission to port P at site K takes place. The SEND site number (and conceivably the SEND port number) is appended to the messages of the transmission for the edification of the receiving process.




Process B may simultaneously wish to execute a RECEIVE from port N at port M.

*Note that there is only one important control message in this system which moves between sites*—the type of message that is called a Host/Host protocol message in [2]. This control message is the RECEIVE message. There are two other possible intersite control messages: (1) a message to the originating site when a RECEIVE or SEND is timed out, and (2) the SEND message in the rare case which we will soon see when the rendezvous site is not the SEND site. There must also be a standard format for messages between ports, for example, the following:

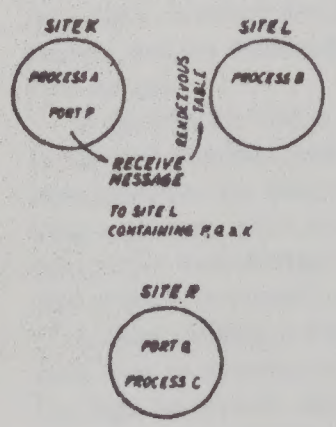



*FOR A FORCE SEND MESSAGE, THE RENDEZVOUS SITE IS THE DESTINATION SITE

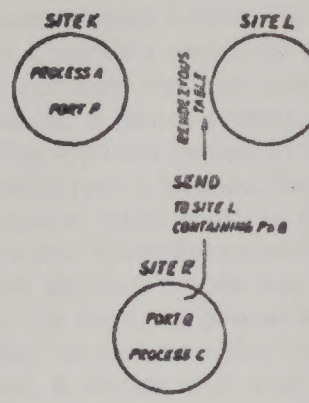
In the model time-sharing system it was possible to pass a port from process to process. This is still possible with a distributed Network Controller.

Remember, that for a message to be sent from one process to another, a SEND and a matching RECEIVE must rendezvous. Both processes keep track of where they think the rendezvous site is and supply this site as a parameter of appropriate operations. The RECEIVE process thinks it is the SEND site and the SEND process normally thinks it is the SEND site also. Since once a SEND and a RECEIVE rendezvous the transmission is sent to the source of the RECEIVE and the entry in the rendezvous table is cleared and must be set up again for each further transmission, it is easy for a RECEIVE port to be moved. If a process sends both the port numbers and the rendezvous site number to a new process at some other site which executes a RECEIVE using these same old port numbers and rendezvous site specification, the SENDer never knows the RECEIVEr has moved.

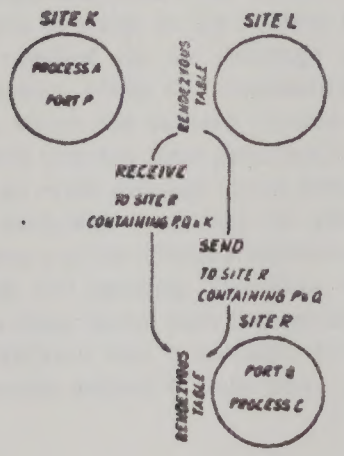
It is slightly harder for a SEND port to be moved. However, if it is, it can be done as follows: Suppose port Q from the preceding example moves from process B at site L to process C at site R. Process A will still think the rendezvous site is site L, and a RECEIVE message will still be sent there.



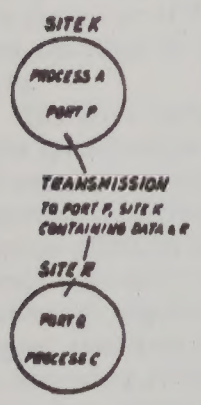
Process C specifies the old rendezvous site L, with the *first* SEND from the new site, R.




When the rendezvous is made at site L, the entry in the rendezvous table is cleared and both the SEND and RECEIVE messages are sent to the source of the SEND message, just as if they had been destined to go there all along.




After the SEND and RECEIVE meet again at the new rendezvous site, R, transmission may continue as if the port had never moved.




Since all transmissions contain the source site number, further RECEIVES will be sent to the new rendezvous site. It is possible to discover that this special manipulation must take place because a SEND message is received at a site that did not originate the SEND message. Note that the SEND port and the RECEIVE port can move concurrently.

Of course, all of this could also have been done if the processes had sent messages back and forth announcing any potential moves and the new site numbers.

A problem that may have occurred to the reader is how the SEND and RECEIVE buffers get matched for size. The easiest solution would be to require that all buffers have a common size, but this is unacceptable since it does not easily extend to a situation where processes in autonomous operating systems are attempting to communicate.

A second solution having great appeal due to its simplicity is for the processes to pass messages specifying buffer sizes. If this solution is adopted, excessive data sent from the SEND process and unable to fit into the RECEIVE buffer is discarded and the RECEIVE process notified.

A third solution would be for the RECEIVE buffer size to be passed to the SEND site with the RECEIVE message and to notify the SEND process when too much data is sent or even to pass the RECEIVE buffer size on to the SEND process. This last method would also permit the Network Controller at the SEND site to make two or more SENDS out of one, if that were necessary to match a small RECEIVE buffer size.

The maintenance of unique numbers is also a problem when the processes are geographically distributed. Of the three solutions to this problem presented here, the first is for the autonomous operating systems to ask the Network Controller for the unique numbers originally and then guarantee the integrity of any unique numbers currently owned by local processes and programs using whatever means are at the operating system's disposal. In this case, the Network Controller would provide a method for a unique number to be sent *from* one site to another and would vouch for the number's identity at the new site.

The second method is simply to give the unique numbers to the processes that are using them, depending on the nonmalicious behavior of the processes to preserve the unique numbers, or if an accident should happen, the *two* passwords (SEND and RECEIVE port numbers) that are required to initiate a transmission. If the unique numbers are given out in a nonsequential manner and are reasonably long (say 32 bits), there is little danger.

In the final method, a user identification is included in the port numbers, and the individual operating systems guarantee the integrity of these identification bits. Thus a process, while not able to be sure that the correct port is transmitting to him, can be sure that some port of the correct user is transmitting.

A third difficult problem arises when remote processes wish to communicate—the problem of maintaining high bandwidth connections between the remote processes. The solution to this problem lies in allowing

the processes considerable information about the state of an ongoing transmission.

First, we examine a SEND process in detail. When a process executes a SEND, the local portion of the Network Controller passes the SEND on to the rendezvous site, normally the local site. When a RECEIVE arrives matching a pending SEND, the Network Controller notifies the SEND process by causing an interrupt to the specified restart location. Simultaneously, the Network Controller starts shipping the SEND buffer to the RECEIVE site. When transmission is complete, a flag is set which the SEND process can test. While a transmission is taking place, the process may ask the Network Controller to perform other operations, including other SENDS. A second SEND over a pair of ports already in the act of transmission is noted, and the SEND becomes active as soon as the first transmission is complete. A third identical SEND results in an error message to the SENDing process.

Next, we examine a RECEIVE process in detail. When a process executes a RECEIVE, the RECEIVE is sent to the rendezvous site. When data resultant from this RECEIVE starts to arrive at the RECEIVE site, the RECEIVE process is notified via an interrupt to the specified restart location. When the transmission is complete, a flag is set which the RECEIVE process can test. A second RECEIVE over the same port pair is allowed. A third results in an error message to the RECEIVE process. Thus there is sufficient machinery to allow a pair of processes always to have both a transmission in progress and the next one pending. Therefore, no efficiency is lost. On the other hand, each transmission must be preceded by a RECEIVE into a specified buffer, thus continuing to provide control of data flow.

## 4. A Hypothetical Application

Only one resource sharing computer network currently exists, the ARPA Computer Network. This section is a discussion of the hypothetical application of the system described in this paper to the ARPA Network [2, 4, 9].

The ARPA Network currently incorporates over 20 sites spread across the United States. Each site consists of one to three (potentially four) independent computer systems called Hosts and one communications computer system called an IMP. All of the Hosts at a site are directly connected to the IMP. The IMPs themselves are connected together by 50-kilobit phone lines (much higher rate lines are a potential), although each IMP is connected to only one to five other IMPs. The IMPs provide a communications subnet through which the Hosts communicate. Data is sent through the communications subnet in messages less than (about) 8,100 bits long. When a message is received by the IMP at the destination site, that IMP sends an acknowledgment, called a RFNM, (Request for Next Message), to the source site. To allow more than one "logical transmission" to be in progress at a time between a particular pair of destination and source Hosts, each message is given a *link number*, and all messages in one logical transmission are sent with the same link number.

A system for interprocess communication for the ARPA Network (let us call this IPC for ARPA) has already been designed by the ARPA Network Working Group, under the chairmanship of S. Crocker of ARPA, and this design is currently being refined and implemented.[7] In contrast to the IPC of this paper, in IPC for ARPA, before two processes can communicate, a *connection* must be set up between a send socket in one process's monitor and a receive socket in the other process's monitor (a *socket* is an element in a network-wide name space into which each monitor maps its own internal port name space). For two processes to make connection, each process makes a connection request to its own monitor. The two monitors then exchange these requests via messages over the *control link*, a special link between each pair of Hosts which is always reserved for control messages. If both monitors are in agreement, the connection is established and a link is assigned for use by the new connection. This new connection exists until it is explicitly terminated, again using intermonitor control messages over the control link. During the "life" of the connection, many messages may be sent from the port at one end to the port at the other, and all these messages are sent over the link assigned to the connection. However, since the connection exists over a series of many messages, a mechanism is provided to stop the flow of messages when the receiving process's Host is overloaded. This mechanism is a system where-by the sender is notified of the receiver's buffering capacity as part of the connection setup, and the sender stops transmitting when it has transmitted enough messages to exhaust the capacity, unless a control message carrying notification of replenishment of the buffering capacity at the receiver arrives first.

The IPC for ARPA comes in several amost distinct pieces: The Host/IMP protocol, the IMP/IMP protocol, and the Host/Host protocol. The IMPs have sole responsibility for correctly transmitting bits from one site to another; the Hosts have sole responsibility for making interprocess connections; both the Host and IMP are concerned and take a little responsibility for flow control and message sequencing.

[7] This work is documented in a series of working group notes which, with the exception of [2], have unfortunately not found their way into the open literature.

[8] This also allows messages to be completely thrown away by the IMP subnet if that should ever be useful.

[9] This is in sharp contrast to IPC for ARPA which seems to utilize software "line switching" on top of a network designed in large part to demonstrate the utility of "message switching."

Application of the IPC described in this paper to the ARPA network suggests a different allocation of responsibility: The IMP still continues to move bits from one site to another correctly, but the Network Controller also resides in the IMP (whereas in the IPC for ARPA it resides in the Host), and flow control is completely in the hands of the processes running in the Hosts, although using the mechanisms provided by the Network Controllers in the IMPS.

The SEND, RECEIVE, FORCE SEND, RECEIVE ANY, and UNIQUE operations are implemented in the IMPS and are available to each IMP's Host. The IMPS also maintain the rendezvous tables, including moving of SEND ports when necessary. Putting these operations in the IMP requires the Host/Host protocol program to be written only once, rather than many times as is currently being done in the ARPA Network—a significant saving notwithstanding the still necessary Host/IMP software interface. It is the author's belief that the existing IMPS could probably be made to include the Network Controller with the addition of 4K of core.

It is perhaps useful to step through the five operations again.

SEND. The Host gives the IMP a SEND port number, a RECEIVE port number, the rendezvous site, and a buffer specification (e.g. start and end, or beginning and length). The SEND is sent to the rendezvous site IMP, normally the local IMP. When a matching RECEIVE arrives at the local IMP, the Host is notified of the RECEIVE port of the just-arrived message. This port number is sufficient to identify the SENDing process, although a given operating system may have to keep internal tables mapping this port number into a useful internal process identifier.

Simultaneously, the IMP begins to ask the Host for specific pieces of the SEND buffer and to send these pieces as network messages to the destination site. If a RFNM is not received for too long, implying a network message has been lost in the network, the Host is asked for the same data again and it is retransmitted.[8] Except for the last piece of a buffer, the IMP requests pieces from the Host which are common multiples of the word size of the source Host, IMP, and destination Host. This avoids mid-transmission word alignment problems.

RECEIVE. The Host gives the IMP a SEND port, a RECEIVE port, a rendezvous site, and a buffer descriptor. The RECEIVE message is sent to the rendezvous site. As the network messages making up a transmission arrive for the RECEIVE port, they are passed to the Host along with the RECEIVE port number (and perhaps the SEND port number), and (using the buffer descriptor) an indication to the Host where to put this data in its input buffer. When the last network message of the SEND buffer is passed into the Host, it is marked accordingly and the Host can then detect this. (It is conceivable that the RECEIVE message could also allocate a piece of network bandwidth while making its network traverse to the rendezvous site.)

RECEIVE ANY. The Host gives the IMP a RECEIVE port and a buffer descriptor. This works the same as RECEIVE but assumes the local site to be the rendezvous site.

FORCE SEND. The Host gives the IMP RECEIVE and SEND ports, the destination site, and a buffer descriptor. The IMP requests and transmits the buffer as fast as possible. A FORCE SEND for a nonexistent port is discarded at the destination site.

In the ARPA Network, the Hosts are required by the IMPS to physically break their transmissions into messages, and successive messages of a single transmission must be delayed until the RFNM is received for the previous message. In the system described here, since RFNMs are tied to the transmission of a particular piece of buffer and since the Hosts allow the IMPS to reassemble buffers in the Hosts by the IMP telling the Host where to put each buffer piece, pieces of a single buffer can be transmitted in parallel network messages and several RFNMs can be outstanding simultaneously. This enables the Hosts to deal with transmissions of more natural sizes and higher bandwidth for a single transmission.

## 5. Conclusion

Since the system described in this paper has not been implemented, there are no clearly demonstrable conclusions and no performance reports that can be presented. Instead, we note what the author regards as the two significant features of the IPC system.

The first significant feature of this IPC is that connections are not maintained over a sequence of messages but instead are set up and expire on a message-by-message basis.[9] There are several advantages to this approach:

1. There is no need for a large number of intermonitor control messages to set up and break connections and consequently no need for a special "out of band" control channel.
2. The RECEIVing process has the opportunity after *each* message to stop the flow of messages. Further, since the RECEIVE process must provide a buffer, the monitor is relieved of the task of providing an indefinitely large buffering capacity.
3. Because connections exist only fleetingly, relatively complex operations such as moving a port are easy.
4. Errors have a minimal effect. For instance, if a

RECEIVE message is lost, the RECEIVing monitor will time it out and the process can do another RECEIVE just as is normally done when the SENDER refuses to SEND in response to a RECEIVE.

5. This IPC is suitable for implementation on small computers as well as large computers due to its simplicity and the nonnecessity for a large buffering capacity in the monitor.

There are also some disadvantages to this IPC; for instance:

1. There is a large amount of overhead associated with each message (e.g. port numbers).
2. To match the speed of a system based on prolonged connections, (e.g. IPC for ARPA) a significant amount of complexity had to be introduced.

The second significant feature of this IPC is the emphasis upon communication between processes and therefore the resources themselves, rather than between monitors. In the model time-sharing system described in Section 2 all possible functions (resources) were given to processes, and in Section 3 the processes (resources) control their own interprocess communication. While this seems to the author to be the proper emphasis if one's goal is flexible resource sharing, it does introduce new problems. For instance, many readers of early versions of this paper have asked, "How does a user who uses a remote process without going through the remote system's logger get billed?" There are ad hoc answers to this and probably to other infirmities in the IPC. (One answer to this particular question is that each process can send off a message to the accounting process whenever it finds itself being used.) However, we conclude with a more general *assertion* about the solutions to such questions.

Elegant solutions to many problems that will arise as computer networks are constructed and their effective utilization attempted will only be found if the computer networks are treated as single entities rather than as associations of autonomous systems. Far from allowing basically incompatible operating systems to communicate, an effective computer network will prove to require similar operating systems throughout the network, albeit running on machines of varying manufacture. General acceptance of this "fact" and its ramifications will bring closer the day when computer networks and resource sharing live up to their potential.

*Acknowledgments.* The author drew on many sources while developing the system suggested in this paper. Particularly influential were: (1) an early sketch of a Host protocol for the ARPA Network by S. Crocker of ARPA and W. Crowther of Bolt Beranek and Newman Inc. (BBN); (2) Ackerman and Plummer's paper [1] on the MIT PDP-1 time-sharing system; and (3) discussions with W. Crowther and R. Kahn of BBN about Host protocol, flow control, and message routing for the ARPA Network. B. Cosell of BBN and the referees made many incisive comments on the presentation of the paper.

Received August 1970; revised March 1971

## References

1. Ackerman, W., and Plummer, W. An implementation of a multi-processing computer system. Proc. ACM Symp. on Operating System Principles, Gatlinsburg, Tenn., Oct. 1–4, 1967.
2. Carr, C., Crocker, S., and Cerf, V. Host/Host communication protocol in the ARPA network. Proc. AFIPS 1970 SJCC., Vol. 36, AFIPS Press, Montvale, N.J., pp. 589–597.
3. Dennis, J., and VanHorn, E. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Mar. 1966), 143–155.
4. Heart, F., Kahn, R., Ornstein, S., Crowther, W., and Walden, D. The interface message processor for the ARPA computer network. Proc., AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J. pp. 551–567.
5. Lampson, B. SDS 940 lectures, circulated informally.
6. Lampson, B. An overview of the CAL time-sharing system. Computer Center, U. of California, Berkeley.
7. Lampson, B. Dynamic protection structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27–38.
8. Roberts, L. The ARPA Computer Network. Networks of Computers Symposium NOC-68. Proc. of Invitational Workshop, Ft. Meade, Md., National Security Agency, Sept. 1969.
9. Roberts, L., and Wessler, B. Computer network development to achieve resource sharing. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 543–549.
10. Spier, M., and Organick, E. The MULTICS interprocess communication facility. Proc. ACM Second Symp. on Operating Systems Principles, Princeton U., Oct. 20–22, 1969.